ScaleG: A Distributed Disk-based System for Vertex-centric Graph Processing

Journal:	*Transactions on Knowledge and Data Engineering*
Manuscript ID	TKDE-2020-08-0960
Manuscript Type:	Regular
Keywords:	Graph Processing, Distributed System, Scalability, Disk I/O

ScaleG: A Distributed Disk-based System for Vertex-centric Graph Processing

Xubo Wang, Dong Wen, Lu Qin, Lijun Chang and Ying Zhang

Abstract—Designing distributed graph systems has drawn a lot of research interests due to the strong expressiveness of the graph model and rapidly increasing graph volume. Most of them require the graph data and all intermediate messages to reside in main memory, which may sacrifice the scalability. Even though several disk-based systems have been studied to remedy such issue, several challenges still exist in achieving both high computational efficiency and low network communication under the limitation of memory usage. In this paper, we design a novel disk-based distributed graph system, called SCALEG. The system provides a series of user-friendly programming interfaces. Unlike previous systems, the programmer in SCALEG does not need to concern any logic regarding the communication between vertices like sending messages and combining messages. In addition to a simple and clear programming model, we propose several techniques to reduce both disk I/Os in each machine and message I/Os via the network. We manage all messages in memory and bound all messages by the number of vertices. We also carefully design the data structure to support partial computation and automatic vertex activation. We conduct extensive experiments on six big graphs to show the high efficiency of our system.

Index Terms—Graph processing, distributed system, scalability, disk I/O

1 INTRODUCTION

GRAPH is a ubiquitous structure representing entities and their relationships. It is applied in many areas such as social network, web graph, road network, and biology. Basic graph problems like pagerank, connected component detection, graph coloring, etc., play fundamental roles in many real-life applications. Efficiently processing graph data is essential in both research and practice. Numerous research interests have been shown on designing distributed graph systems to process big graphs [1]–[9].

The vertex-centric programming model, initially proposed by Pregel [6], requires programmers to provide the behavior of each vertex in developing distributed graph algorithms. Pregel adopts a Bulk Synchronous Parallel (BSP) model, and the deployed algorithm runs in several iterations. Two key functions in Pregel are `Compute` and `SendMsg`. `Compute` is implemented by the programmer for the logic to manipulate each vertex. `SendMsg` is invoked by the programmer to send customized messages to neighbors of the vertex. The vertex-centric model is user-friendly and naturally captures the characteristics of many fundamental graph problems like pagerank and graph coloring. Given the advent of Pregel [6], various following vertex-centric systems have been proposed, such as PowerGraph [10], Pregel+ [7], and Giraph [4]. Some of them adopt the similar push strategy, where each vertex takes control of sending messages but passively receives messages. Others adopt a pull logic, where each vertex actively requests information from neighbors.

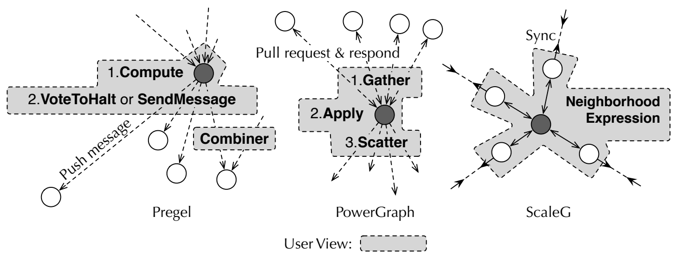


Fig. 1. Representative system models

Most distributed vertex-centric graph systems store all data in main memory of machines, which brings high efficiency but sacrifices scalability given the dramatic increasing data volume. Due to some intermediate results, messages, replicated vertices and edges, the memory usage can be much larger than the input graph size. To remedy the scalability issue, we aim to design a new disk-based distributed graph systems for implementing efficient and scalable vertex-centric graph algorithms. Several disk-based (or called out-of-core) distributed graph systems have been studied in the literature, including Pregelix [11], Chaos [12] and GraphD [13]. Among them, GraphD is the state-of-the-art and follows the same programming model as Pregel [6]. GraphD adopts the semi-streaming model and only allows the vertex states resided in main memory of each machine. The adjacency lists and messages are managed as edge streams and message streams on disks, respectively. Several optimizations are proposed to achieve high I/O efficiency in scanning vertex neighbors, sending and receiving messages.

Motivation. There are still several challenges in GraphD. First, under the memory usage limitation, GraphD saves all

- *Xubo Wangis with the University of Sydney, Australia. Email: xubo.wang@sydney.edu.au.*
- *Dong Wen is with the University of Technology Sydney, Australia. Email: dong.wen@uts.edu.au.*
- *Lu Qin is with the University of Technology Sydney, Australia. Email: lu.qin@uts.edu.au.*
- *Lijun Chang is with the University of Sydney, Australia. Email: lijun.chang@sydney.edu.au.*
- *Ying Zhang is with the University of Technology Sydney, Australia. Email: ying.zhang@uts.edu.au.*

sending messages and receiving messages on disks. Scanning and managing the message streams on disks incurs a great deal of disk I/Os. Second, several studies [7], [9], [14] have shown that the volume of communication messages can be very large given the power-law degree distributions of real-world graphs [15]. To handle such issue, the pull-based computing model is studied in the literature [9], [14]. However, disk I/O is not considered in these in-memory systems. More importantly, the pull-based method requires an extra pull request to notify the corresponding neighbor. In addition, based on the push model of Pregel, GraphD cannot efficiently handle the partial computation in many fundamental algorithms. For example, in the algorithm of distributed graph coloring, only a small number of vertices are active and wait to be colored in many iterations. To color a vertex u, we need colors of all neighbors of u. Since each vertex can only passively receive messages in Pregel-like systems, all vertices have to send messages to their neighbors. Therefore, the main challenges in this paper are how to effectively reduce the communication messages and how to efficiently manage the messages under the limitation of memory usage.

Our Approach. In response to the above challenges, we propose an efficient disk-based distributed graph system, called ScaleG, with a series of simple and user-friendly programming interfaces. We observe that vertices in many vertex-centric algorithms only communicate with neighbors. We enforce such property in the system and adopt a compute-and-sync programming model. The compute phase performs the logic provided by the user, while the sync phase synchronizes the vertex states in different machines and is hidden from users. Thanks to our computing model, all neighbors' states are locally provided in the compute phase. The programmer only needs to care about how to update the vertex based on neighbors' states and does not need to concern any logic regarding message sending, receiving or combining. An illustration of our model is given in Fig. 1, with Pregel and PowerGraph as comparisons.

To implement the computing model in ScaleG, we use the same semi-streaming model as GraphD to store all edges on disk. However, unlike GraphD, we maintain all messages in the main memory. The rationale of in-memory message management is supported by our computing model, which bounds the number of messages by the number of vertices in each machine. In addition, we propose detailed external-memory data structures for vertex activation. Unlike GraphD, the messages with the same value would never be sent repeatedly in all studied algorithms in ScaleG. For example, in the implementation of the graph coloring algorithm in ScaleG, colors of all neighbors can be locally accessed by each vertex, and in following iterations, only the changed colors will lead to an update message.

Contribution. We summarize the main contributions in this paper as follows.

- *An elegant out-of-core distributed graph systems.* We design a new disk-based distributed graph system, called ScaleG, where the programmer does not need to concern any behavior regarding sending messages, receiving messages, and combining messages.
- *In-memory message management.* With our computing model, ScaleG manages all intermediate messages in memory, which avoids numerous disk I/O cost. The total disk I/O of all machines in each iteration is bounded by $O(m/B)$ where m and B stand for the number of edges and block size respectively.
- *Efficient partial computation.* Under the limitation of memory usage, ScaleG efficiently activates a partial set of vertices with high I/O efficiency and avoids any unnecessary message transmission.
- *Extensive performance studies.* We implement nine fundamental and various distributed graph algorithms on six large real-world graphs. We conduct extensive experiments compared with several representative competitors to show the outperformance of ScaleG.

Organization. The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 presents the computing model of our system. Section 4 gives the implementation details and Section 5 lists a few algorithm designs. Section 6 reports the performance studies, and Section 7 concludes the paper.

2 Preliminary and Related Work

Graph processing system has been studied intensively. We mainly review representative systems that are related to our work here.

2.1 Distributed Vertex-centric System

In-memory Pregel [6] is the first vertex-centric in-memory graph processing system utilizing the iterative properties of many graph algorithms. It adopts the bulk synchronous parallel (BSP) model [16] which consists of iterations. Inside each iteration, vertices conduct the user defined function which includes value computation as well as communication with other vertices. Giraph [17] is an open-source implementation of Pregel in Java. GPS [18] presents an optimization technique, large adjacency list partitioning, for high-degree vertices. PowerGraph (GraphLab) [5], [9] adopts the vertex-cut partition schema and supports both synchronous and asynchronous computation modes. It adopts a Gather, Apply, and Scatter (GAS) programming model where users still think like a vertex. Yan et al. [7] designed a system named Pregel+ implementing Pregel with message reduction and load balancing techniques. Chen et al. [14] adopt the GAS model from PowerGraph and design a differentiated processing model based on vertex degree. Zhu et al. [19] propose an adaptive switching model depending on the density of active edges in different applications. In this work, we focus on out-of-core systems considering system scalability.

Out-of-core Because distributed in-memory systems provide high efficiency but are weak in scalability, some distributed external-memory systems are proposed to compensate [11]–[13]. Pregelix [11] implements the Pregel programming model with an iterative dataflow of relational operators like join and group-by and supports both in-memory and out-of-core workloads. Chaos [12] is a distributed version of a single-PC out-of-core system X-stream which sequantially scans all edges. It is inefficient especially for sparse computation where the number of active vertex is small and inactive edges could be skipped [13]. Besides, Chaos is built for the assumption that a cluster is connected by high-speed network and streaming data from a remote device is acceptable. Its performance is undesirable for Gigabit Ethernet [12], [13] which is used by many clusters. Yan et al. propose a distributed out-of-core graph system GraphD [13] based on a semi-streaming model where vertex states are stored in memory, and edges and messages are streamed from the disk. The adopted push-based model of GraphD generates large volume of messages which leads to intense communication cost and disk I/O.

2.2 Other Systems

Single-machine Single-machine graph processing systems store and process a given graph in a single machine. Existing systems include in-memory systems like Ligra [20] and Galois [21], [22] and out-of-core systems like GraphChi [23], TurboGraph [24], VENUS [25] and so on [26]–[29]. Single-machine graph processing systems have high efficiency because of communication cost saving and fast convergence. However, the disadvantage is weak scalability due to limited hardware resources. Considering system scalability, in this paper, we aim at a distributed graph processing system which can utilize resources of all machines in a cluster.

Subgraph-centric There is another category of graph processing systems that allows users to program with a subgraph [3], [30]–[35]. Yan et al. [35] designed Blogel where each connected subgraph is a block and users program functions for blocks. NScale [31] and Arabesque [32] adopt the k-hop neighborhood-centric model based on MapReduce framework. TurboGraph++ [34] supports k-hop neighborhood centric analysis and extends a single out-of-core graph processing system TurboGraph [24] to a distributed environment. G-Miner [30] models subgraph mining problems as independent tasks and provides a task-based pipeline to asynchronously process CPU, Network, Disk I/O operations for efficiency.

General Optimization There are also some studies on general graph processing system optimization techniques [1], [4], [36]–[42]. Salihoglu et al. [38] propose some optimization techniques to implement algorithms efficiently on Pregel-like systems. Considering the traditional push and request-respond pull mechanisms generate a large number of messages, Wang et al. [40] designed an automatic switching mechanism between push and pull models to optimize system performances. Song et al. [39] put forward a redundancy reduction strategy to achieve high-performance graph analytics by taking advantage of graph structure. The other works focus on improving system efficiency through new hardwares, like SSDs, GPUs [37], [41], [42]. We leave these optimization works out of comparison in our study.

3 SCALEG ABSTRACTION

Given a graph $G(V, E)$, where V is the set of vertices and E is the set of edges, we use n and m to denote $|V|$ and $|E|$, respectively. In this paper, we assume that the graph is undirected for ease of presentation. The proposed ideas and techniques can be easily extended on directed graphs, which is discussed in Section 5. For a vertex $u \in V$, $N(u)$ denotes neighbors of u in undirected graphs, and $deg(u)$ denotes $|N(u)|$. We use $\{\mathcal{W}_1, \mathcal{W}_2, ..., \mathcal{W}_k\}$ to denote a cluster of working machines, where k is the number of machines.

3.1 Challenges in Existing Out-of-Core Systems

Most vertex-centric systems adopt either a push-based or a pull-based method in message transmission between vertices. A summary can be found in [40].

As a Pregel-like system, GraphD takes a push-based logic in each iteration. Specifically, a vertex in Pregel-like systems sends messages to other vertices voluntarily and executes the computing function only based on the received messages in the last iteration. The push-based method may incur large communication cost and generate a large volume of messages in the receiver machine since each active vertex sends messages to all neighbors in each iteration. Due to the limited memory resources, an external-memory buffer (e.g., message streams in GraphD [13]) is required in each machine to manage the messages, which brings extra disk I/Os to operate the buffer. If the message values are associative and commutative, a combiner optimization can be used to reduce messages according to their destination vertices in sender machines [6]. However, the combiner only works on specific algorithms such as PageRank and SSSP, but not available in algorithms like core decomposition and graph coloring. We will show the details in Section 5. Even though the messages can be combined, the degree distribution is skewed in real-world power-law graphs, and high-degree vertices still generate many messages [9]. Some systems [4], [18] even do not combine messages due to the poor locality of destination vertices.

To overcome the drawbacks in push logic, several in-memory systems adopt a pull-based method. We take PowerGraph [9] as a representation. Each vertex requests all necessary neighbor values in the computing function. Given limited memory resource, the pull-based method outperforms the push-based method when the message volume is large [40]. However, disk I/O is not studied in these in-memory systems. More importantly, under pull logic, each vertex requires an extra pull request to derive the neighbor values, which brings considerable communication cost.

Simply caching all in-neighbor values for each vertex takes considerable memory space and cannot reside in main memory which incurs many disk I/Os.

3.2 Motivation of Our Approach

Given the challenges discussed in Section 3.1, we design a new distributed graph processing system called SCALEG. Following most of existing vertex-centric systems, we adopt the Bulk Synchronous Parallel (BSP) model. Specifically, each algorithm deployed on SCALEG runs in several iterations (or called supersteps in some papers). In each iteration, a set of active vertices perform their own computing function assigned by users. The algorithm terminates if there is no active vertex.

SCALEG outperforms existing disk-based systems in the following three aspects. First, regarding the communication cost, SCALEG not only alleviates the great amount of sending messages incurred by high-degree vertices in push-based methods but also avoids the extra pull requests in pull-based methods. Second, unlike GraphD [13], SCALEG bounds the size of communication messages and always maintains the sending and receiving messages in memory, which significantly speed up the local computation in each machine. Third, users in ScaleG only need to care the computing logic for each vertex. All other tasks including vertex interactions (e.g., *SendMessage* in Pregel [6], *Gather* and *Scatter* in PowerGraph [9]) and message reductions (e.g., *Combiner* in Pregel [6], *Mirror* in Pregel+ [7]) will be handled by ScaleG automatically and invisibly.

To achieve these goals, we observe several common characteristics of the algorithms studied in the papers of existing vertex-centric systems. We make two assumptions for the algorithms deployed on ScaleG as follows.

Assumption 1. *The computation of each vertex only depends on its neighbors.*

In other words, Assumption 1 means that a vertex cannot communicate with any non-neighbor vertex. Without loss of generality, we treat all variables accessed in the deployed algorithm as attributes of the vertex. The attributes include basic vertex structural properties (e.g. ID and degree) and algorithm-specific values (e.g. PageRank value in PageRank and core number in core decomposition).

Assumption 2. *There exists one or more attributes $\mathbb{A} = \{\mathcal{A}_1, \mathcal{A}_2, ...\}$ for each vertex such that an arbitrary vertex u is active in the i-th iteration iff $\exists \mathcal{A} \in \mathbb{A}, v \in N(u)$, the attribute $\mathcal{A}$ of v changed in the $(i-1)$-th iteration.*

```
void Compute(Vertex u, Message msgs):
  int dist = IsSource(u)? 0 : INF;
  for(msg in msgs):
    dist = min(dist,msg.val);
  if(dist < u.val){
    u.val = dist;
    for(v in u.nbrs){
      SendMsg(v, dist+1);
    }
  }
  voteToHalt();

Message combine(Message msgs):
  int dist = INF;
  for(msg in msgs):
    dist = min(dist,msg.val);
  return Message(dist);
```

Snippet 1. **SSSP in GraphD and other Pregel-like Systems**

Example 1. *We give an example to explain Assumption 1 and Assumption 2. Snippet 1 gives an implementation of the single-source shortest path (SSSP) algorithm in GraphD and other Pregel-like systems. In the algorithm, each vertex updates the shortest distance value only depending on its neighbors' values. We regard the distance value (`dist` in the snippet) as an attribute of each vertex in this case. We can find that a vertex u is active if the attribute value of any neighbor of u changed in last iteration. Similarly, the attribute value of each vertex in PageRank, graph coloring, and core decomposition are PageRank value, color ID, and core number, respectively.*

Our two assumptions regarding the deployed algorithm do limit the system flexibility to some extent. However, in addition to the SSSP algorithm in Example 1, almost all other algorithms studied in existing vertex-centric systems naturally match the assumptions. The algorithms include but are not limited to Pagerank, detecting connected components, graph coloring, core decomposition, maximal matching, maximal independent set and triangle counting. We implement these algorithms on our system, and the experimental details can be found in Section 6.

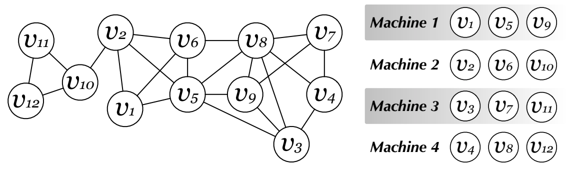

v_{11}
v_2
v_6
v_8
v_7
v_{10}
v_{12}
v_1
v_5
v_9
v_4
v_3
Machine 1 v_1 v_5 v_9
Machine 2 v_2 v_6 v_{10}
Machine 3 v_3 v_7 v_{11}
Machine 4 v_4 v_8 v_{12}

Fig. 2. A demo graph and the vertex partition

3.3 Execution Model and Programming API

ScaleG adopts a compute and sync vertex program in each iteration. Fig. 1 shows the execution model of ScaleG, with those of Pregel and PowerGraph as comparisons.

3.3.1 *Computing via Neighborhood Expression*

In the compute phase, based on Assumption 1, the ScaleG abstraction gives users the permission to locally access all neighbors for each vertex. As a result, from the user's perspective, only the logic to process a vertex with all given neighbors is required when deploying algorithms in ScaleG, and we call the logic *Neighborhood Expression*.

Remark. *We are not the first to provide all neighbors' attributes (values) for each vertex in the programming model. Distributed GraphLab [5] also uses a similar programming abstraction and hides all network communication from the user. However, unlike our system, Distributed GraphLab adopts the asynchronous execution model in system design and maintains all data in memory of each machine. While, in our paper, we propose a disk-based system in BSP execution, which focuses on the efficiency and the scalability of processing large graphs given the limited memory resource. Compared with Distributed GraphLab, our main technical contributions center on designing the data structure for several key components like message sending/receiving, and automatic vertex activation, under the limitation of our programming model and system setting.*

3.3.2 *Sync via Attributes*

The sync phase synchronizes the vertex states in all working machines and guarantees that the states of all neighbors are up-to-date for each vertex in the next iteration. Unlike previous vertex-centric BSP systems, the sync phase and all other jobs, like message sending and vertex activation, are hidden from users and managed automatically by ScaleG. The viability of this idea is supported by Assumption 2. Specifically, users are required to assign one or more attributes for the vertex. The assigned attribute is monitored by the system and plays an important role in two aspects. First, the attributes of each vertex will be synchronized among all machines if the vertex copies are stored in multiple machines. The details regarding the distributed graph organization will be given in Section 4.1. Only the changed attributes are transferred between machines in each iteration. Second, once any attribute of a vertex u changes, all neighbors of u will be activated by the system in next iteration. The system will terminate the algorithm if there is no active vertex.

```
void Attr(attributeName, attributeType[, value]);
/*assign an attribute*/
void Exec(function(Vertex[, List<Vertex>]){
/*execute the neighborhood expression*/
...
}[, iterNum]);
```

Snippet 2. **ScaleG Programming Interface**

3.3.3 *System API*

The main API offered by ScaleG is summarized in Snippet 2. There are only two key functions. `Attr()` is used to assign an attribute for the vertex. The third parameter `value` is optional to assign a default value to the attribute. `Attr()` can be invoked repeatedly to add multiple attributes. `Exec()` is used to execute the neighborhood expression given in the first parameter, which is an interface implemented by users. As earlier discussed, the user can locally access all neighbors for each vertex. Therefore, the interface exposes two parameters to users — a vertex and all its neighbors grouped in a list.

```
1 Attr("dist", int, INF);
2 Exec(function(u,nbrs){
3   if(IsSource(u)) u.dist = 0;
4   for(v in nbrs){
5     u.dist = min(u.dist,v.dist+1);
6   }
7 });
```

Snippet 3. **Single Source Shortest Path (BFS) in ScaleG**

BFS in ScaleG. The implementation of BFS algorithm in ScaleG is given in Snippet 3. The first line sets an attribute called *dist* for the vertex and initializes *dist* of each vertex as infinity. In line 2, `u` is the current processing vertex, and `nbrs` is the neighbor list of `u`. Lines 4–6 update the attribute `dist` of u based on the `dist` value of each neighbor. Note that there is no code in Snippet 3 related to the logic of communication such as sending messages, receiving messages and activating vertices. Initially, all vertices are activated by the system. Assume that the source vertex is u_0. Only the `dist` value of u_0 changes from `INF` to 0 in the first iteration. Given the attribute `dist`, the system first synchronizes the `dist` value of u_0 in all machines if necessary. Then in the second iteration, the system automatically activates all neighbors of u_0 since the `dist` value of u_0 changes. The following iterations will perform the same strategy, and the system terminates the procedure automatically if the `dist` values of all vertices do not change.

4 Implementation

4.1 Distributed Graph Organization

We partition vertices in the graph to different working machines in a hashing way, which is same as many systems like GraphD and Pregel. We leave the integration of ScaleG with other partition methods like Metis [43] and load balancing techniques from Mizan [44] and GPS [18] out of this work considering the effectiveness may be limited [45]. A demo graph and corresponding vertex partition are given in Fig. 2.

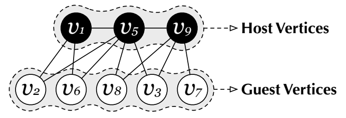


Fig. 3. The graph structure in machine 1 based on the partition in Fig. 2

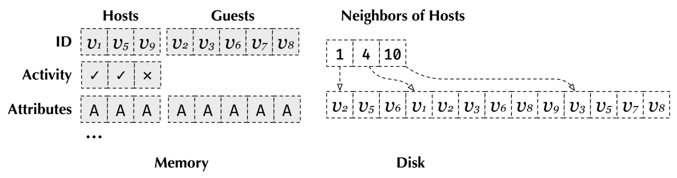

Hosts
Guests
Neighbors of Hosts
ID
v_1 v_5 v_9
v_2 v_3 v_6 v_7 v_8
Activity
✓ ✓ ×
Attributes
A A A
A A A A A
...
1 4 10
v_2 v_5 v_6 v_1 v_2 v_3 v_6 v_8 v_9 v_3 v_5 v_7 v_8
Memory
Disk

Fig. 4. The data structure for the compute phase in machine 1

Given a working machine $\mathcal{W}$, the vertices assigned to $\mathcal{W}$ by the hash function are called *host vertices* for $\mathcal{W}$, denoted as $V(\mathcal{W})$. The *guest vertices* for $\mathcal{W}$, denoted by $N(\mathcal{W})$, are the neighbors of boundary vertices in $\mathcal{W}$, i.e., $N(\mathcal{W}) = \{u \notin V(\mathcal{W}) | \exists v \in V(\mathcal{W}), u \in N(v)\}$. An example for host vertices and guest vertices are given in Fig. 3.

Similar to GraphD, ScaleG adopts the semi-external setting and a compressed sparse row (CSR) structure to maintain the states of host vertices in memory and store the neighbors of each vertex on the disk. Unlike GraphD, ScaleG additionally maintains the guest vertices in the memory of each machine to support the locally neighborhood expression for the host vertices. Even considering both host vertices and guest vertices, the size is acceptable and still significantly smaller than that of all edges. The space complexity of memory usage will be given in Theorem 3.

Example 2. *The data structure regarding the neighborhood expression in machine 1 of Fig. 2 is given in Fig. 4. The corresponding subgraph visualization can be found in Fig. 3. In memory, in addition to the vertices' IDs, we have a bitmap to maintain whether each vertex is active or not and a set of attribute arrays assigned by the users. In the SSSP algorithm, an attribute array maintains the shortest distance for each vertex to the source vertex. The neighbors of each host vertices are shown on the right of Fig. 4. The first array is an index file, and for each vertex, the file stores the starting position for the vertex's neighbors in the neighbor list file. The neighbor list file stores neighbors of all vertices sequentially. To derive all neighbors of a given vertex u, we derive the start position of the neighbors of u in the index file, and the end position is the next value of u in the index file. For instance, given a vertex v_5, the start position and the end position are* 4 *and* 10, *respectively.*

In the compute phase of each iteration, ScaleG first sequentially scans the activity bitmap. Once meeting an active vertex u, we jump to the starting point of the neighbors of u in the neighbor list and retrieve all neighbors of u from the disk. Then, the system invokes the `exec` function and updates the attributes of u based on the neighborhood expression defined by the user. Note that the attributes of all neighbors of u can be accessed in memory. In each iteration, we only sequentially read or skip items in the neighbor list file from disk. We have the following disk I/O complexity.

Theorem 1. *In each iteration of* ScaleG, *the total disk I/O of all machines is bounded by $O(m/B)$, where B is the block size*

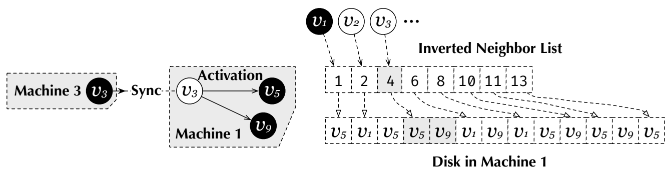


Fig. 5. The data structure for the sync phase in machine 1

for a single disk read/write operation.

Note that GraphD performs an external sort to organize the messages on the disks, which cannot bound the same disk I/O as our result.

4.2 In-Memory Message Organization

This subsection introduces the details regarding sending and receiving messages in the sync phase. Unlike GraphD, messages in ScaleG are always organized in memory, since we can bound the messages by the number of vertices in each machine.

Sending Messages. In the main memory of each machine, we have a sending buffer for every other machine. For each vertex u, a bitmap structure records all machines in which u is a guest vertex. During the computation, if any attribute value of u changes, we add the new attribute value of u to the sending buffers of all corresponding machines by checking the bitmap. Note that the messages in the sending buffer of each machine are naturally arranged in increasing order of their IDs due to our scheme of sequential vertex processing. The memory usage for sending buffers and bitmaps in each machine are bounded by $O(k \cdot |V(\mathcal{W})|)$ and $O(|V(\mathcal{W})|)$, respectively. Vertex u will be marked back to $inactive$ for the next iteration after the sending information is buffered. The system will clear the sending buffer after all messages are sent for synchronization.

Receiving Messages. In each machine, instead of using one buffer to receive and sort all messages, we have $k - 1$ receiving buffers corresponding to $k - 1$ other machines. The rationale is that the received vertices in each buffer are already in increasing order of their IDs. Based on this property, we can efficiently update guest vertex attributes and activate host vertices, which will be introduced in Section 4.3 in detail. The memory bound of the receiving buffer is $O(N(\mathcal{W}))$.

Based on the above discussion, the communication cost of ScaleG is analyzed as follows.

Theorem 2. *In each iteration of* ScaleG, *the communication cost of all machines is bounded by* $O(\min(n \cdot k, m))$.

Associated with the memory usage for the compute phase in Section 4.1, we bound the overall memory usage of each machine in ScaleG as follows.

Theorem 3. *In each iteration of* ScaleG, *the memory usage of an arbitrary machine* $\mathcal{W}$ *is bounded by* $O(k \cdot V(\mathcal{W}) + N(\mathcal{W}))$.

4.3 Vertex Activation

This subsection introduces details of the vertex activation process in each machine after messages are received.

Inverted Neighbor List. The system activates host vertices based on a data structure called *Inverted Neighbor List* on the disk. For each vertex u in a working machine $\mathcal{W}$, the inverted neighbors of u, denoted by $I_{\mathcal{W}}(u)$, are the neighbors of u in the host vertices of $\mathcal{W}$, i.e., $I_{\mathcal{W}}(u) = N(u) \cap V(\mathcal{W})$. Note that the inverted neighbors of a vertex u are not the neighbors in the induced subgraph of all vertices in the machine, not even neighbors in the whole graph. Similar to the structure of the neighbor list in Section 4.1, we use a data file to store the inverted neighbors of all vertices in a sequence and an index file to store the starting position of the inverted neighbors of each vertex in the data file. Following the example of Fig. 2, the inverted neighbor list in machine 1 is presented on the right of Fig. 5. For example, the inverted neighbors of v_2 are located from the second position in the inverted neighbor list which are v_1 and v_5.

Recall that we have receiving buffers containing updated guest vertices from $k - 1$ other machines. In these buffers, messages are in increasing order of vertex IDs. An extra list is utilized to contain updated host vertices on this machine. Then, we conduct a k-way-merge-like process on the receiving buffers and the list to sequentially update guest vertices and activate host vertices. Specifically, we select the vertex u with the least ID from the k buffers and list each time. Then, we jump to the starting position of the inverted neighbors of u. We load all inverted neighbors of u from the disk and mark all of them as $active$. If u is a guest vertex (from the receiving buffers), we update its local attributes accordingly. Since we process the vertices in increasing order of their IDs, the activation process only sequentially scans the inverted neighbor list once in each iteration. Therefore, the disk I/O cost in the activation process is loosely bounded by $O(m)$, and Theorem 1 still holds.

Example 3. *We give an example to illustrate the process of synchronizing attributes and activating vertices on the left of Fig. 5. Assume that a vertex v_3 updates an attribute value in machine 3. By checking the neighbors of v_3 in machine 3, we know that v_3 is a guest vertex in machine 1. Then machine 3 sends a message to machine 1 with the updated attribute of the vertex v_3. After receiving the message, machine 1 first updates the attribute value of the guest vertex v_3 and then load the inverted neighbors of v_3. The items related to the vertex v_3 in the list are marked by gray on the right. We derive the inverted neighbors v_5 and v_9. Finally, the system marks v_5 and v_9 as $active$.*

Adaptive Activation. We also design an adaptive activation mechanism to boost the activation efficiency. We consider the cases where most vertices change their values like the earlier iterations of detecting connected components and graph coloring. In these cases, the inverted neighbors of updated vertices are likely to cover all vertices in the graph. Thus, instead of scanning inverted neighbor lists, we directly activate all vertices when the number of updated vertices exceeds a threshold. When less vertices update their values, the activation mechanism is automatically switched to the normal activation process based on the inverted neighbor list. The number of updated vertices is recorded to implement our idea. We set the threshold as $n/50$ which empirically shows a good performance in our tests.

Handling Directed Graphs. For directed graphs, we maintain both in-neighbors and out-neighbors of each host vertex (Section 4.1). When a vertex reads in-neighbors to compute the attribute, the in-neighbors of each host vertex are provided for the neighborhood expression, and the out-neighbors are used for sending message to corresponding machines. In this case, the inverted neighbors list (Section 4.3) maintains the inverted host out-neighbors, which is used to activate host vertices. We also have a separate file maintaining the inverted host in-neighbors of each vertex to support some vertex-centric algorithms which compute vertex attributes based on the out-neighbors, even though such algorithms are few.

5 Algorithm Case Studies

Due to the space limit, we only give the examples of PageRank, graph coloring, and maximal matching implemented by ScaleG in this section.

```
Attr("pr", double, 1/n);
Attr("deg", int, GetDegree());
Exec(function(u,nbrs){
  u.pr = 0.15/n;
  for(v in nbrs){
    u.pr += 0.85*v.pr/v.deg;
  }
}, TargetNum);
```

Snippet 4. **PageRank in ScaleG**

PageRank. The implementation of distributed PageRank algorithm is presented in Snippet 4. Two attributes are assigned to each vertex. pr represents the PageRank value and is updated in each iteration. deg is just used for the PageRank computation. The function `GetDegree()` scans the disk file and derives the degree of each vertex. The algorithm will run `TargetNum` iterations. Note that ScaleG also supports terminating the algorithms when the PageRank value is less than a given threshold by overriding the equality function of the "pr" attribute.

```
Attr("color", int);
Attr("deg", int, GetDegree());
Exec(function(u,nbrs){
  for(v in nbrs: v.deg > u.deg || {v.deg == u.deg
      && v.id>u.id}){
    if(v.color is undefined) return;
    mark v.color as used;
  }
  u.color = 0;
  while(u.color is used){
    u.color = u.color+1;
  }
});
```

Snippet 5. **Greedy Graph Coloring in ScaleG**

Graph Coloring. The implementation of distributed graph coloring algorithm is presented in Snippet 5. The algorithm colors vertices in non-increasing order of vertex degrees and breaks the tie by the vertex ID. In the initial iteration, all vertices are active, and the vertex u with the largest degree is assigned by 0. Then, in the second iteration, the color of u is synchronized among all machines, and the neighbors of u are activated. We can see that for each vertex u, the message with a color number is sent only once from u to other machines with u as a guest during the algorithm.

Maximal Matching. The implementation of distributed maximal matching algorithm [46] is presented in Snippet 6. In the algorithm, a vertex has more than one attributes to sync which are "pick" and "match" here. In ScaleG, no attribute transmission will occur when the picked vertex remains the same as in the previous round. By contrast, all attributes need to be sent in GraphD in this case.

TABLE 1
Characteristics of datasets

Dataset	$\|V\|$	$\|E\|$	deg_{max}	deg_{avg}
DB	986,207	13,414,472	979	13.60
OR	2,997,167	212,698,418	27,466	70.97
UK	18,520,343	523,574,516	194,955	28.27
TW	41,652,230	2,936,729,768	2,997,487	70.51
FR	65,608,366	3,612,134,270	5,214	28.93
CW	978,409,098	42,574,107,469	75,611,696	43.51

```
Attr("pick", int, -1);
Attr("match", int, -1);
Attr("deg", int, GetDegree());
Exec(function(u,nbrs){
if(u.match < 0 && u.deg > 0 && u.pick != -2){
if(iter_num % 2 == 0){
u.pick == -1;
for(v in nbrs: v.match < 0 && v.id > u.pick){
u.pick = v.id;
}
if(u.pick == -1) u.pick = -2;
} else{
for(v in nbrs: v.pick == u.id && u.pick == v.id){
u.match == v.id; break;
}
}
});
```

Snippet 6. **Maximal Matching in ScaleG**

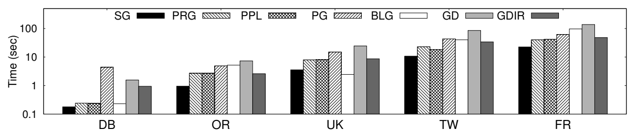
6 Experiments

Datasets. We use 6 real-world datasets of different sizes obtained from LAW [47]. DBLP (DB), Orkut (OR), Twitter (TW) and Friendster (FR) are social network graphs. UK and ClueWeb (CW) are webgraphs. Table 1 shows the dataset details. $|V|$ and $|E|$ represent the number of vertices and edges respectively. deg_{max} and deg_{avg} denote the maximum and average vertex degree in each dataset respectively.

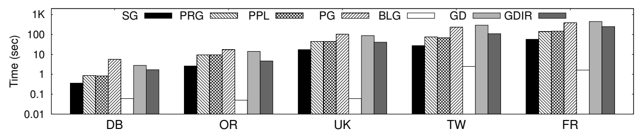
Experimental settings. We run our experiments on a cluster of 10 machines connected by Gigabit Ethernet. Each machine has one 3.0GHz Intel Xeon E3-1120 CPU (4 cores), 64GB DDR3 RAM and 610GB disk. Unless specified, we use 6 machines, each with 4 cores by default.

We compare our system ScaleG with 5 representative existing systems including in-memory systems: Pregel, Pregel+ [48], PowerGraph [10], and out-of-core system GraphD [49]. We also include Blogel [50] for some researchers' interests [35], [51]. All systems are implemented in C++. We use Yan's implentation [48] of Pregel. In terms of Pregel+, we adopt the mirroring mode in the experiments. Similar to [48], we select the vertex mirror threshold as the minimum value between 1000 and the value computed using their cost model. If not stated, we use the default settings of compared systems. For ease of expression, we represent the system names ScaleG, Pregel, Pregel+, PowerGraph, Blogel and GraphD by SG, PRG, PPL, PG, BLG respectively. GD and GDIR represent GraphD without and with its ID recoding technique, respectively.

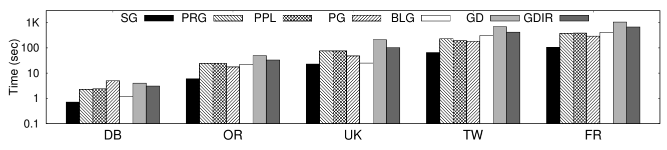
Algorithms. To evaluate system performance, we use 9 algorithms including single-phase algorithms: breadth first search (BFS), connected component (CC), PageRank (PR), personalized PageRank (PPR), core decomposition (Core)



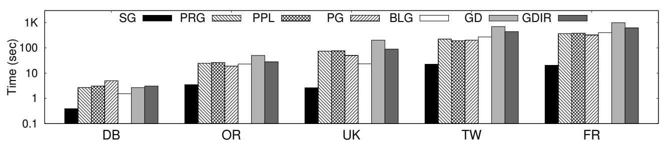
(a) BFS



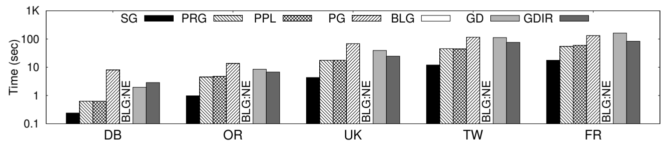
(b) CC



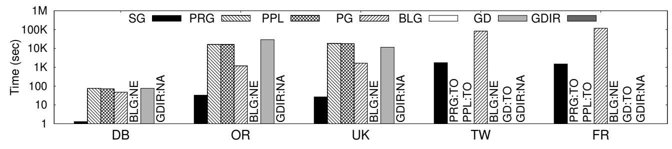
(c) PR



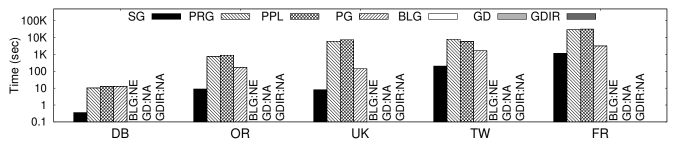
(d) PPR



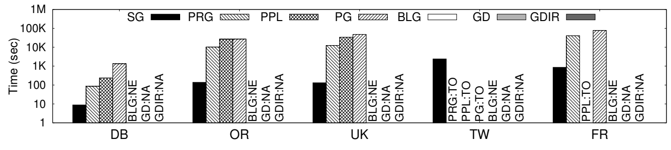
(e) MIS



(f) Color



(g) Core



(h) MM

Fig. 6. Comparison with Existing Systems (Running Time)

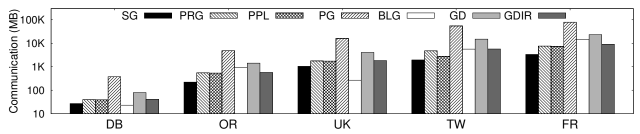
[52] and graph coloring (Color) [53] and multi-phase algorithms: maximal independent set (MIS) [54], maximal matching (MM) [53]. Among them, BFS, CC, PR, PPR and MIS are separable algorithms. Core, Color and MM are non-separable algorithms. An algorithm is separable if commutative and associative operation is to be applied on transmitted messages where optimization techniques like combiner can be applied. The ID recoding of GraphD is also only applicable to separable algorithms. We also include triangle counting (TC) for scalability testing. Due to the space limit, we omit the implementation details here. Note that BFS, CC and PR are the most popular algorithms that existing works adopt to test system performance, and they are all single-phase separable algorithms. To the best of our knowledge, this is the first work to involve such various algorithms to show the stability of system performance.

Metrics. We report the *running time* and *communication cost* to compare the system performance. *Running time* is counted from the moment when the data graph is totally loaded in the cluster to the time when the computation is completed. Note that data loading and result dumping time are excluded. *Communication cost* is the sum of data size transferred among workers in the cluster. Note that neither the cost of partitioning an input graph nor distributing it to workers is included. To specially compare with the disk-based system GraphD, we also report *disk I/O* and *memory cost* in Section 6.3.

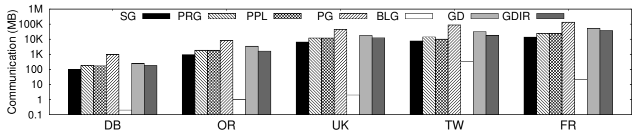
6.1 Efficiency over Different Algorithms

We compare the system efficiency when running different algorithms over given datasets. The running time results are shown in Fig. 6. We use NE and NA to represent the cases that the system is Not Effective or Not Applicable to that algorithm respectively. OOM, OOD and TO represent Out Of Memory, Out Of Disk and Time Out respectively. We consider an algorithm running as time out when it can't finish within 24 hours. We can see that SG exhibits the best overall performance over different algorithms.

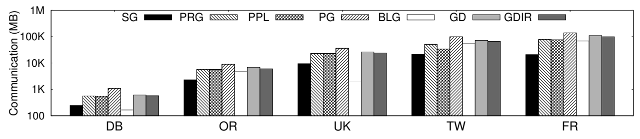
Fig. 6 (a)-(e) show the running times on separable algorithms which are popularly adopted for comparison in existing works. SG runs significantly and consistently faster than disk-based system GraphD. Specifically, SG outperforms GD and GDIR by 14.7x and 7.8x on average respectively. This benefits from no message disk I/O and less communication cost in SG. The attributes syncing of SG avoids unnecessary message transmission which not only reduces communication cost but also supports keeping messages in memory. However, the push-based method adopted in GraphD causes large volume of message transmission which causes high communication cost (see Fig. 7). To guarantee scalability, GraphD saves messages on disk which, however, sacrifices efficiency because of high message I/O cost. Besides, merge-sorting message files on disk in GraphD is very time-consuming. The ID recoding technique of GDIR improves efficiency compared to GD by eliminating incoming message disk I/O. However, the outgoing message disk I/O is still unavoidable. SG also outperforms in-memory systems PRG, PPL and PG by 5.2x, 5.3x and 9.8x on average respectively. This is because SG saves communication cost from the message sending incurred by high-degree vertices in push-based methods (PRG and PPL) and extra pull requests in pull-based methods (PG). SG is outperformed by the



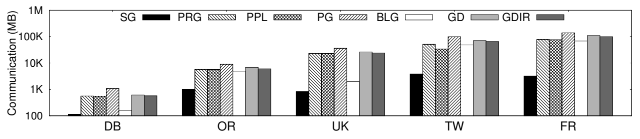
(a) BFS



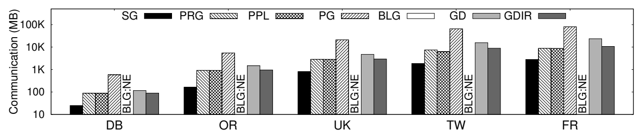
(b) CC



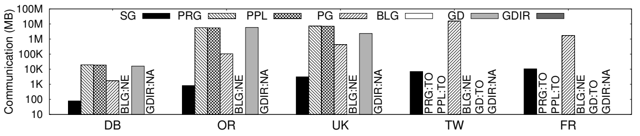
(c) PR



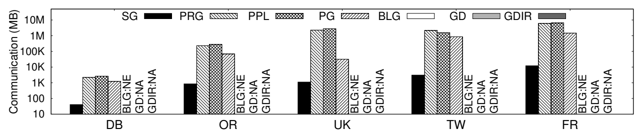
(d) PPR



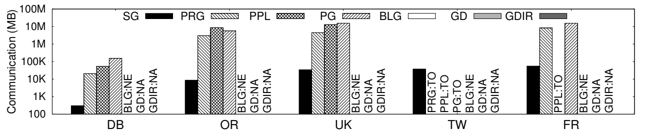
(e) MIS



(f) Color



(g) Core



(h) MM

Fig. 7. Comparison with Existing Systems (Communication Cost)

block-centric system BLG running CC because BLG saves communication cost in a subgraph. However, BLG is only effective on specific algorithms where vertices in the same subgraph share the same values like CC [40].

Fig. 6 (f)-(h) show the experimental results on non-separable algorithms. The advantages of SG over these systems are still considerable. For example, SG outperforms PRG, PPL, PG, and GD by 198.2x, 249.3x, 89.4x, and 311.8x on average, respectively. The speedup can even reach 906.0x when running Core on UK compared to PPL. This is because the optimization techniques in existing systems like combiners, ID recoding which are suppose to reduce communication cost and disk I/O cost are inapplicable on non-separable algorithms.

Communication Cost Comparison We also report the communication cost comparison results of evaluated systems in Fig. 7. The results are mostly consistent with the running time presented above considering more communication cost leads to longer running time. In most cases, SG incurs less communication cost than that of compared systems due to the sync model. The gap is even bigger on non-separable algorithms where optimization techniques like combiners are inapplicable. BLG has less communication cost in PR and PPR because we only report the communication cost of B-mode [50] (same with running time). The whole program needs to run V-mode first.

6.2 Scalability Test

In this section, we evaluate the scalability of all systems by varying the number of tested graph size and used machines respectively. We choose 5 representative algorithms BFS, PR, Core, MM and TC to report the results in this part.

Varying the Number of Machines. Firstly, we test the scalability of SG in comparison with existing systems by varying the number of used machines. For each machine, all four cores are used. We run selected algorithms over two large datasets Twitter and Friendster. The results are shown in Fig. 8.

The experimental results show that, in most cases, with the increasing number of machines used, better efficiencies are achieved. This is because the greater number of machines used, more parallel computation happens and the less computation time consumes. As a result, total running time reduces. However, more machines also means more communication cost. So, when saved computation cost doesn't compensate increased communication cost, the total time couldn't be reduced but will increase. This explains in some cases how the more machines used, more time is consumed. For example, running Core on PG over Twitter, the total time increases when eight machines used compared to six machines.

Among existing systems, in-memory systems show better efficiency than disk-based system GD. However, for memory-intensive algorithm like TC which generates huge amount of messages, all in-memory systems cause OOM error. While GD can finish running TC when all 10 machines are used. Note that when less machines are used for GD, the large volume of messages causes OOD error. Different from existing systems, SG shows excellent overall performance for all kinds of algorithms. In terms of cpu-intensive algorithms, it outperforms existing in-memory systems. For example, it is averagely 36.6, 27.1 and 11.4 times faster than PRG, PPL and PG respectively running Core on Twitter for different number of machines. For memory-intensive

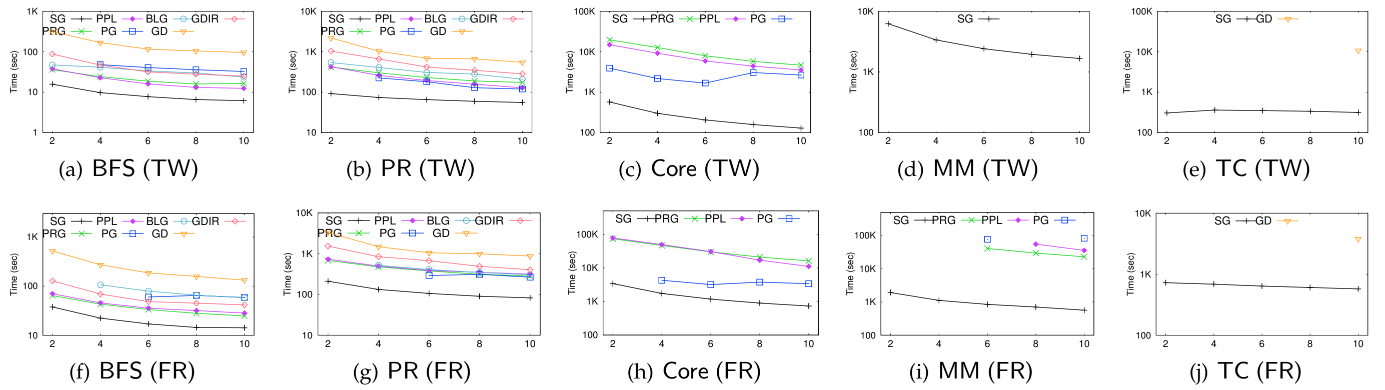


Fig. 8. Scalability Test (Varying #machines)

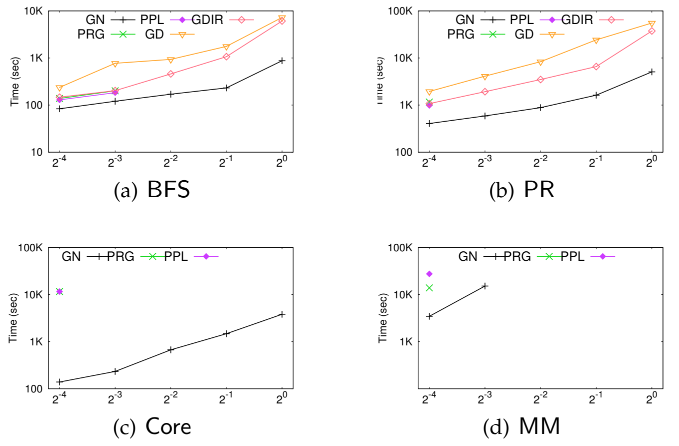


Fig. 9. Scalability Test (Varying #edges)

algorithms, SG is competitive compared to GD. For instance, SG and GD are the only two systems that finish running TC on Twitter. It is worth noticing that GD can only finish when all ten machines are used. Nevertheless, SG is able to finish even when only two machines are used which makes our system more generally applicable. Adding to this, though both systems finish when all 10 machines are used, SG is 33.1 times faster than GD. This further demonstrates the good combinational performance of SG compared with existing systems.

Varying Graph Size. We also test the system scalability by varying graph size. We adopt the largest used dataset ClueWeb and randomly sample 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4} of all edges to vary the graph size. The experimental results are shown in Fig. 9. Note that TC results are not reported because no system can finish running TC on any used dataset within 24 hours. Also, PG and BLG are not shown because they both couldn't finish on used big graphs because of OOM error.

The results show that with dataset size increasing, the running time of all systems increase as well. All systems show similar increasing behavior. In-memory systems PRG and PPL show good efficiency but can't finish when the graph is too large. For example, they can only finish on the smallest graph used when running PR. While disk-based system GraphD shows better scalability but weak efficiency. GD is the slowest system in the results. With ID recoding, GDIR gets similar running time with in-memory systems PRG and PPL. Our new system SG shows better performance in terms of efficiency and scalability. For example, only SG and GraphD finish running BFS on the largest used graph within 24 hours. Besides, SG is 7.3 and 6.0 times faster than GD and GDIR respectively.

6.3 Comparison with GraphD

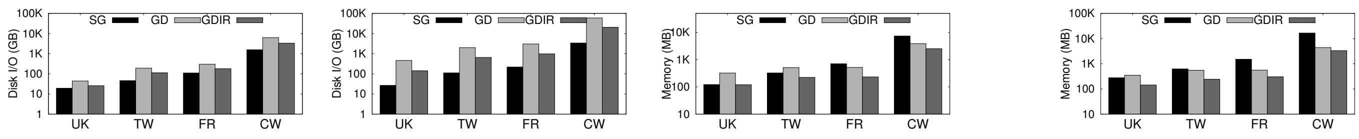
We also compare our system with GD in terms of disk I/O and memory cost. In terms of disk I/O, we report the total disk input and output of all machines in the cluster. In terms of the memory cost, we report the maximum resident memory usage across all machines used in the experiments. Due to the space limit, we only report the results of running BFS and PR over four largest datasets UK, TW, FR and CW in Fig. 10.

Disk I/O. From Fig. 10(a) and Fig. 10(b) , we can see that the disk I/O of GraphD is much larger than that of ScaleG. This is because the large amounts of messages in GraphD incur a lot of disk read and write operations for storing generated messages from memory into disk and loading stored messages for sending from disk to memory. In particular, the average disk I/Os of GD are 3.6 and 16.4 times that of SG for BFS and PR respectively. We can also find that ID recoding is effective on reducing messages hence reduces disk I/O. As a result, the average disk I/Os of GDIR are 2.1 and 5.4 times that of SG for BFS and PR respectively. However, for non-seperable applications where combiners and ID recoding are not applicable, the differences are more severe. For example, the disk I/Os of GD running Color on OR and UK are 58.9 and 24.1 times of that of SG respectively.

Memory. In terms of memory cost, we can see that the memory cost of SG is generally higher than GraphD because messages are kept on disk for GraphD. However, even in a small cluster like used in our experiment, SG can run applications on graphs with billion edges. The fact that GD requires extra memory buffers for message streams and merge-sorting compared to SG explains why on smaller datasets like UK, GD shows a bit larger memory cost than SG. ID recoding in GDIR helps to reduce memory cost.

6.4 Other Issues

Fault tolerance is important to a system. It is not considered in current work because the authors of PowerGraph state in their paper [9] that the overhead, typically a few seconds for



(a) Disk I/O (BFS) (b) Disk I/O (PR) (c) Memory (BFS) (d) Memory (PR)

Fig. 10. Comparison with GraphD

largest graph used, is relatively small compared to the total running time. This is consistent with our experiments. For example, the total time of PG for running Color on Twitter, a dataset also used in their paper, is 23 hours. However, we leave the implementation of SG's fault tolerance in the future. Also, asynchronous mode is not considered because it is not general and is only effective on algorithms with asymmetric convergence behavior and low workload [45].

7 Conclusion

We propose a disk-based system ScaleG for vertex-centric graph processing with a simple programming interface in this work. Programmers only need to specify the computing logic. Several techniques are proposed to reduce both disk I/Os in each machine and message I/Os via the network. In ScaleG, all messages are managed in memory and are bounded by the number of vertices. Disk I/O is bounded by $O(m/B)$ where m and B stand for the number of edges and block size respectively. Different structures are carefully designed so that ScaleG efficiently supports partial computation and automatic vertex activation. Extensive experimental results validate the superb efficiency of ScaleG on large graphs.

Acknowledgment

Lu Qin is supported by ARC FT200100787. Lijun Chang is supported by ARC DP160101513 and FT180100256. Ying Zhang is supported by ARC DP180103096 and FT170100128.

References

[1] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 752–768.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian, "Parallelizing sequential graph computations," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 495–510.

[4] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[5] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[7] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1307–1317.

[8] Q. Zhang, A. Acharya, H. Chen, S. Arora, A. Chen, V. Liu, and B. T. Loo, "Optimizing declarative graph queries at large scale," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 1411–1428.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[10] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Powergraph," 2010. [Online]. Available: https://github.com/jegonzal/PowerGraph

[11] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big (ger) graph analytics on a dataflow engine," *Proceedings of the VLDB Endowment*, vol. 8, no. 2, pp. 161–172, 2014.

[12] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 410–424.

[13] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang, "Graphd: distributed vertex-centric graph processing beyond the memory limit," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 99–114, 2018.

[14] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, L. Réveillère, T. Harris, and M. Herlihy, Eds. ACM, 2015, pp. 1:1–1:15. [Online]. Available: https://doi.org/10.1145/2741948.2741970

[15] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM*, L. Chapin, J. P. G. Sterbenz, G. M. Parulkar, and J. S. Turner, Eds. ACM, 1999, pp. 251–262.

[16] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[17] giraph.apache.org, "Giraph," 2011. [Online]. Available: http://giraph.apache.org/

[18] S. Salihoglu and J. Widom, "Gps: a graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.

[19] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system." in *OSDI*, 2016, pp. 301–316.

[20] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211–222, 2007.

[22] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 456–471.

[23] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc." USENIX, 2012.

[24] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 77–85.

[25] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He, "Venus: Vertex-centric streamlined graph computation on a single pc," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1131–1142.

[26] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "Nxgraph: An efficient graph processing system on a single machine," in

Data Engineering (ICDE), 2016 IEEE 32nd International Conference on. IEEE, 2016, pp. 409–420.

[27] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.

[28] K. Vora, G. H. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing." in *USENIX Annual Technical Conference*, 2016, pp. 507–522.

[29] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, vol. 13, 2013, pp. 3–6.

[30] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: an efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 32.

[31] A. Quamar, A. Deshpande, and J. Lin, "Nscale: neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 25, no. 2, pp. 125–150, 2016.

[32] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 425–440.

[33] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[34] S. Ko and W.-S. Han, "Turbograph++: A scalable and fast graph analytics system," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 395–410.

[35] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[36] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang, "Efficient distributed memory management with rdma and caching," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1604–1617, 2018.

[37] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, ser. FAST'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 285–299. [Online]. Available: http://dl.acm.org/citation.cfm?id=3129633.3129659

[38] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 577–588, 2014.

[39] S. Song, X. Liu, Q. Wu, A. Gerstlauer, T. Li, and L. K. John, "Start late or finish early: A distributed graph processing system with redundancy reduction," *PVLDB*, vol. 12, no. 2, pp. 154–168, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol12/p154-song.pdf

[40] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 479–494.

[41] D. M. Da Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 45–58.

[42] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[43] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[44] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 169–182.

[45] D. Yan, Y. Bu, Y. Tian, and A. Deshpande, "Big graph analytics platforms," *Found. Trends Databases*, vol. 7, no. 1-2, pp. 1–195, 2017. [Online]. Available: https://doi.org/10.1561/1900000056

[46] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 553–564.

[47] law.di.unimi.it, "The laboratory for web algorithmics," 2002. [Online]. Available: http://law.di.unimi.it/

[48] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Pregel+," 2015. [Online]. Available: http://www.cse.cuhk.edu.hk/pregelplus/

[49] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang, "Graphd," 2018. [Online]. Available: http://www.cse.cuhk.edu.hk/systems/graphd/

[50] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel," 2014. [Online]. Available: http://www.cse.cuhk.edu.hk/blogel/

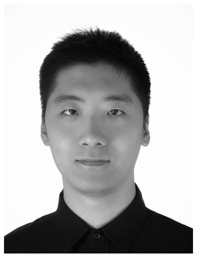
[51] K. Ammar and M. T. Özsu, "Experimental analysis of distributed graph systems," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1151–1164, 2018.

[52] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2012.

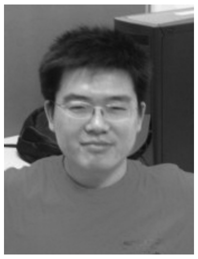
[53] N. Linial, "Locality in distributed graph algorithms," *SIAM Journal on Computing*, vol. 21, no. 1, pp. 193–201, 1992.

[54] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

Xubo Wang reveived his BEng and MEng degree from Xiamen University, and PhD degree from the University of New South Wales in 2011, 2014 and 2018 respectively. He is currently a postdoctoral research fellow in the School of Computer Science at the University of Sydney. His research interests include efficient query processing and mining on big graphs.



Dong Wen reveived the BEng degree from Nankai University in 2013, and the PhD degree from the Faculty of Engineering and Information Technology, University of Technology Sydney in 2019. He is currently a postdoctoral research fellow in the Centre for Artificial Intelligence, University of Technology Sydney. His research interests include I/O efficient graph processing and streaming graph analysis.



Lu Qin received the BEng degree from the Department of Computer Science and Technology, Renmin University of China, in 2006, and the PhD degree from the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, in 2010. He is currently an associate professor with the Centre for Artificial Intelligence, University of Technology, Sydney. His research interests include big graph analytics and graph query processing.

Lijun Chang is a Senior Lecturer and ARC Future Fellow in the School of Computer Science at the University of Sydney. He received Bachelor degree from Renmin University of China in 2007, and Ph.D. degree from The Chinese University of Hong Kong in 2011. He worked as a Postdoc and then DECRA research fellow at the University of New South Wales from 2012 to 2017. His research interests are in the fields of big graph (network) analytics, with a focus on designing practical algorithms and developing theoretical foundations for massive graph analysis. He has co-authored two monographs, and published over 50 papers in top venues such as SIGMOD, KDD, PVLDB, ICDE, VLDB Journal, TKDE, and Algorithmica.

Ying Zhang is a professor and ARC Future Fellow at CAI, the University of Technology Sydney (UTS). He received his BSc and MSc degrees in Computer Science from Peking University, and PhD in Computer Science from the University of New South Wales. His research interests include query processing on data stream, uncertain data and graphs. He was an Australian Research Council Australian Postdoctoral Fellowship (ARC APD) holder (2010 – 2013) and ARC DECRA research fellow (2014 – 2016).