Efficient Disk-Based Directed Graph Processing: A Strongly Connected Component Approach

Yu Zhang, Xiaofei Liao, *Member, IEEE*, Xiang Shi, Hai Jin, *Senior Member, IEEE*, and Bingsheng He

Abstract—Recently, there have been many disk-based systems proposed for iterative graph processing. In the popular vertex/edge-centric systems, an iterative directed graph algorithm needs to reprocess many partitions so as to update their vertices' states according to other non-convergent vertices for the unawareness of their dependencies. As a result, it induces high data access cost and a long time to converge. To tackle this problem, we propose a novel system for iterative directed graph processing with taking advantage of the strongly connected component (SCC) structure. It stores a directed graph into a directed acyclic graph (DAG) sketch, with each node representing a SCC in the original data graph. During execution, the SCCs are loaded into memory for processing in a parallel way according to the topological order of the DAG sketch, and the vertices in each SCC are tried to be handled along the directed paths. In this way, each SCC is able to reach convergence in order and needs to be loaded into the main memory for exactly once, getting much lower data access cost and faster convergence. Besides, the vertices of each SCC need fewer updates for convergence. We further develop a lightweight approach to maintain the DAG sketch and handle SCCs in an incremental way for evolving graphs. Compared with the state-of-the-art methods, experimental results show that our approach achieves a performance improvements of 1.46-8.37 times for static graphs, and can reduce the execution time by 61.4-72.7 percent for evolving graphs.

Index Terms—Iterative direct graph processing, strongly connected component, convergence, I/O cost

1 Introduction

Directed graphs, e.g., web graph, biological network and social network, are prevalent in the real world. Many iterative graph algorithms are proposed to analyze them, which are employed in many applications ranging from targeted advertising [1] to language processing [2]. For example, PageRank [3] is employed to rank pages in the World Wide Web. Other examples include Adsorption [4], SSSP [5] and K-Core [6] on many different directed graph networks. They need to process the graph round by round until the results are convergent, i.e., all vertices' states have met a given condition. The performance of these iterative directed graph algorithms has been an important and fruitful research topic.

Despite the increase of memory size per machine, disk-based graph processing system has been a cost-effective approach to achieve acceptable performance. Many disk-based systems, such as GraphChi [7], X-Stream [8], TurboGraph [9], VENUS [10], GridGraph [11] and NXgraph [12], have been proposed for efficient graph processing on a single PC. The key optimization of those systems is to divide the graph into multiple partitions (so that each partition can fit into the main memory), and to accelerate the data access and the convergence speed of each partition.

The disk-based graph processing systems mainly use vertex-centric [7], [9], [10] or edge-centric [8], [11], [12], [13] execution models. They iteratively update each vertex state according to the states of its neighbors until its state is convergent. However, these models are ignorant of the update dependencies between the vertices of a directed graph. A non-convergent vertex may slowly send its new state to its directly/indirectly neighbors and trigger them to be processed repeatedly (causing excessive I/Os), even after several rounds. As a result, for iterative directed graph processing, many graph partitions may need to be reprocessed several rounds for convergence, even when a single vertex is not convergent. It leads to exceptionally high data access cost and slow convergence speed. This paper investigates whether and how we can reduce such redundant I/O accesses and improve the convergence speed.

We observe that, in iterative directed graph processing, a vertex needs fewer state updates if the states of its precursors are more quickly propagated to it along directed paths, and even only needs a single state update when it is processed after the convergence of all its precursors. This motivates us to explore a special substructure in a graph—strongly connected component (SCC), which represents a maximal subgraph in which every pair of nodes is reachable from each other. By contracting each SCC to a node, a directed graph can be translated into a DAG of SCCs (We call it *DAG sketch*).

Based on these observations, we propose a SCC-centric execution model to efficiently support the iterative processing of a directed graph in disk-based systems. It stores a

- *Y. Zhang, X. Liao, X. Shi, and H. Jin are with the Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: zhang_yu9068@163.com, {xfliao, hjin}@hust.edu.cn, 895770932@qq.com.*
- *B. He is with the Department of Computer Science, National University of Singapore, Singapore 117418. E-mail: hebs@comp.nus.edu.sg.*

Manuscript received 31 Mar. 2017; revised 7 Oct. 2017; accepted 13 Oct. 2017. Date of publication 22 Nov. 2017; date of current version 9 Mar. 2018. (Corresponding author: Xiaofei Liao.)
Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2776115

TABLE 1
Summary of Typical Graph Processing Frameworks

Systems	Execution model	Partition scheme	Processing way	Evolving graph
Pregel [14]	Vertex-centric	Edge-cut	Synchronous	Not support
Blogel [15]	Vertex-centric	Edge-cut	Synchronous	Not support
GraphLab [16]	Vertex-centric	Edge-cut	Synchronous/Asynchronous	Not support
PowerGraph [17]	Edge-centric	Vertex-cut	Synchronous/Asynchronous	Not support
Chronos [18], [19]	Vertex-centric	Edge-cut	Synchronous/Asynchronous	Support
GraphChi [7]	Vertex-centric	Edge-cut	Synchronous/Asynchronous	Support
VENUS [10]	Vertex-centric	Edge-cut	Synchronous/Asynchronous	Not support
X-stream [8]	Edge-centric	Vertex-cut	Synchronous	Support
GridGraph [11]	Edge-centric	Vertex-cut	Synchronous/Asynchronous	Not support
PathGraph [13], [20]	Edge-centric	Vertex-cut	Synchronous/Asynchronous	Not support
NXgraph [12]	Edge-centric	Vertex-cut	Synchronous	Not support
DGraph	SCC-centric	Vertex-cut	Synchronous/Asynchronous	Support

directed graph using a DAG sketch of SCCs for efficient graph processing. Whenever the graph needs to be analyzed, its SCCs are loaded and processed in a parallel way according to the topological order of this DAG. Then the processed SCC will never receive vertex states from the other SCCs. Besides, it employs a novel asynchronous processing way to try to handle the vertices in each SCC along the directed paths, so as to maximize the performance. In order to support evolving graphs, it also employs an efficient algorithm to maintain the DAG sketch and process SCCs in an incremental way.

Compared with existing execution models, our SCC-centric graph execution model has two main advantages. (1) It has faster convergence speed for fewer repeated vertex state updates. (2) It has smaller data access cost because it never needs to process a SCC once it has been convergent and also only needs to access a few SCCs for exactly once when processing a SCC. To demonstrate the efficiency of our model, a graph processing system, i.e., DGraph,[1] has been designed and built. Experimental results show that, for static directed graph analysis, DGraph reduces the number of vertex state updates by 33.2-91.0 percent in comparison with a cutting-edge graph processing model [13], [20], and therefore achieves performance improvements of 1.46-8.37 times. Compared with ODS [21], GridGraph [11] and NXgraph [12], three other latest graph processing systems on a single PC, DGraph offers dramatic improvements of 4.7-20.5, 3.9-26.9 and 2.06-4.31 times, respectively. In addition, for an evolving graph with an update ratio of 2-16 percent, DGraph is able to reduce the execution time up to 61.4-72.7 percent in comparison with Chronos [18], [19], a state-of-the-art system for evolving graph analysis.

The remainder of this paper is organized as follows: Section 2 gives a survey of related work and a description of the challenges and our motivation. Section 3 presents the main idea and details of our approach, followed by an experimental evaluation in Section 4. Finally, we conclude this paper in Section 5.

2 Background and Motivation

This section first gives a summary of related work then discusses the challenges and our motivation.

1. https://github.com/CGCL-codes/DGraph

2.1 Related Work

With the explosion of graph's scale, the efficiency of graph processing has attracted lots of attentions [22]. Table 1 gives a summary of the typical systems supporting graph analysis. Generally, graph processing systems can be categorized into distributed systems [23], [24], [25] and disk-based single-machine systems [26], [27].

2.1.1 Distributed Graph Processing Systems

Pregel [14] is one of the earliest distributed systems for graph processing and its graph algorithm (like PageRank) is usually expressed as multiple iterations. For higher computation to data access ratio, Blogel [15] takes a connected subgraph as an unit for processing and iteratively processes the vertices in each subgraph. Zhou et al. [28] uses a geo-aware graph partitioning method to reduce communication cost over geo-distributed platform. Although such a synchronous execution model is simple and easy to verify the correctness of graph algorithms, these systems encounter high synchronization overhead [29], [30] and are inefficient for state propagation among vertices in large graphs.

GraphLab [16] supports the execution of graph algorithm in an asynchronous way. It breaks the global barrier and allows the latest state of each vertex to be immediately used by its neighbors. Such an asynchronous model often has a faster convergence speed and lower synchronization cost than the synchronous model. However, it is still suffer from load imbalance caused by the skew of vertex degrees. Based on GraphLab, PowerGraph [17] evenly divides the whole graph edges into several partitions and proposes to factor vertex-programs over edges for balanced load. As a further optimization, PowerSwitch [31] proposes a hybrid execution mode that allows dynamic switching between synchronous and asynchronous modes to gain optimal performance. Kineograph [32] employs an incremental graph-computation engine for evolving graph analysis to keep up with continuous updates on the graph. Chronos [18], [19], as a state-of-the-art specific framework for evolving graph analysis, uses a locality-aware batch scheduling scheme to maximize the benefit of data locality. However, they still need to process several locally convergent partitions for the unawareness of their dependencies.

2.1.2 Disk-Based Single-Machine Systems

Compared with distributed graph processing systems, disk-based single-machine systems can support large-scale graph

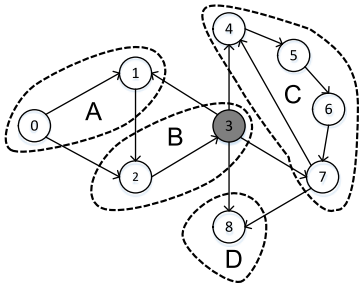


Fig. 1. Inefficiency of current execution models for the iterative processing of directed graph. (The current models still need to process the whole graph several rounds although only the vertex v_3 is not convergent).

algorithm in a comparable time with lighter communication cost and lower power consumption. In this paper, we focus on this kind of graph processing systems for their cost-effectiveness [26], [27]. Current disk-based graph processing systems mainly adopt two execution models, i.e., vertex-centric and edge-centric. However, these methods are still inefficient to iterative directed graph analysis with the performance deficiency in redundant I/O accesses and convergence speed, as described in Section 2.2.

Vertex-Centric Systems. Many graph processing systems are based on vertex-centric model and take the vertex as the processing unit. GraphChi [7] is a typical one and proposes a parallel sliding windows method to efficiently support static/evolving graph processing on PC via reducing the random access cost. However, for GraphChi, all edges in an interval need to be loaded into memory before calculation, resulting in unnecessary disk data transfer. To tackle this problem, TurboGraph [9] proposes pin-and-slide model, which also helps exploiting locality. To reduce data access cost and minimize random I/Os, VENUS [10] proposes an extra data structure, v-shards, to enable streamlined disk access pattern. However, all these systems use static partitions and need to load many edges with no impact on the computation of vertex values. In order to remove this I/O inefficiency, Vora et al. [21] propose an optimization that dynamically changes the layout of a partition structure.

Edge-Centric Systems. On the other hand, compared with the vertex-centric solutions, edge-centric execution model explicitly factors computation over edges instead of vertices and takes the edge as the processing unit, in order to get balanced load, efficient access of edges and so on. X-Stream [8] is a typical one based on edge-centric model. According to the directed edge in streaming partitions, it processes its source vertex and updates its destination vertex for sequential access of edges. In this way, it also can easily handle growing graph. Meanwhile, some studies [33] show that there often is little computation for each vertex in graph algorithms, leading to a memory wall that limits their speedup. Therefore, GridGraph [11] proposes a 2-Level hierarchical partitioning scheme along with a novel dual sliding windows method to apply on-the-fly vertex updates, so as to reduce unnecessary I/O cost. PathGraph [13], [20] models a large graph using a collection of tree-based partitions and iteratively processes each partition, aiming to improve the locality. NXgraph [12] proposes destination-sorted subshard structure to store a graph so as to further ensure locality of graph data access.

2.1.3 Other Related Work on Leveraging SCCs

Recently, some methods are proposed to use connectivity properties to reduce the number of visited vertices for specific algorithms. For example, the algorithms [34], [35] are designed to optimize the calculation of betweenness centrality. Some solutions [36], [37] build a DAG for directed graph and translate reachability query for two vertices on directed graph to the reachability query problem of the SCCs containing these two vertices, so as to reduce the problem size. The work [38] uses an optimization technique based on SCCs to compress a large directed graph for pattern matching queries. Compared with them, our approach uses the DAG sketch to get the dependencies between SCCs so as to reduce the redundant processing of vertices for iterative directed graph processing. Brinkmeier [39] uses a similar approach as ours in order to reduce redundant processing for PageRank algorithm. However, this method suffers from slow vertex state propagation, poor locality and low parallelism for inefficient SCC-level parallel processing and inefficient partitioning and processing of each SCC, especially for the giant SCC.

Meanwhile, some solutions [40], [41] are proposed to divide graph into SCCs. However, with these approaches, the vertices of directed path in a SCC may be divided into many partitions and also stored in a partition in an arbitrary order, incurring slow convergence of this SCC. In contrast, our approach not only generates the DAG sketch but also divides each SCC according to directed paths, so as to allow efficient state propagation along the directed paths and also to improve the locality.

2.2 Challenges and Motivation

In the traditional iterative execution models, i.e., vertex-centric or edge-centric, graph algorithm is often performed via iteratively executing a large number of user-defined functions to update the state of each vertex according to its neighbors until its state has met a given condition. However, with these models, a non-convergent vertex may slowly send its new state to the other processed vertices and cause the reprocessing of many partitions. Therefore, for iterative directed graph algorithms, they may face redundant data accesses and inefficiency in convergence speed.

Let us describe this issue with a simple example. In Fig. 1, the graph is divided into four partitions, i.e., A, B, C and D, and there is only one non-convergent vertex, i.e., v_3, in the current iteration. After the processing of partition B, v_3 sends its new state to v_1, v_4, v_7 and v_8. Then the partitions A, C and D need to be reprocessed to update the states of these vertices, i.e., v_1, v_4, v_7 and v_8, according to the new state of v_3, although these partitions may have been handled. These vertices may also cause their neighbors on other partitions to be non-convergent in a similar way, inducing many partitions to be reprocessed.

Furthermore, with the current synchronous or asynchronous processing way, the vertices' states cannot be efficiently propagated along directed paths. A partition may need to be reprocessed several rounds to update its locally convergent vertices according to slowly propagated states from its other non-convergent vertices, exacerbating the above challenge.

Take the partition C as an example. With the synchronous way [14], each vertex state cannot be used by the vertices in the same iteration. The vertex state can only be

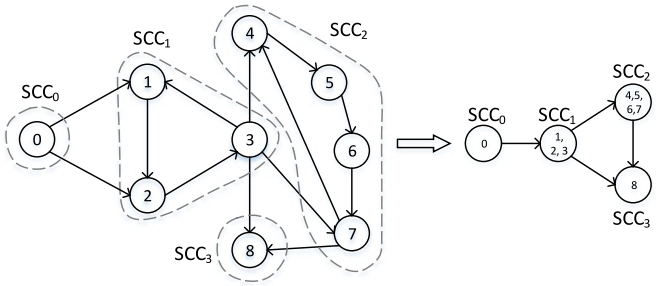


Fig. 2. An example to illustrate how to contract graph to DAG sketch.

propagated to its directed neighbors after one round processing. For convergence, it needs three rounds to process such a partition so as to propagate the state of v_4 to v_7. In the asynchronous way [42], the latest state of each vertex is allowed to be immediately used by its neighbors and is propagated in a faster way than the synchronous way. However, it also may face inefficient state propagation because the vertices may not be handled along the directed path. For example, the vertices in partition C may be processed along the order of v_7, v_6, v_5 and v_4. Then v_7, v_6 and v_5 have been handled before receiving new state of v_4 within the current round. Because the states of already-processed vertices can only be updated in the next round, v_7, v_6 and v_5 have to be reprocessed in the next several rounds to update their states according to the state of v_4. At the least, partition C needs three rounds of processing for convergence because there are three hops for the state propagation from v_4 to v_7.

From the above discussion, one important observation is that, a vertex needs fewer state updates if the states of its precursors are more quickly propagated to it along directed paths, and even only needs a single state update if it is processed after the convergence of all its precursors. This observation motivates us to avoid the redundant processing of partitions for iterative directed graph algorithms via arranging the processing order of vertices.

3 Our SCC-Centric Execution Model

To reduce the redundant I/O accesses and improve the convergence speed, we take advantage of a special substructure in the directed graph called strongly connected component, which represents a maximal subgraph in which every pair of nodes is reachable from each other. Thus, we propose a SCC-centric execution model to efficiently support iterative directed graph analysis. It contracts each SCC to a node so as to represent a directed graph into a DAG (we call it *DAG sketch*). With that transformation, it does not need the reprocessing of convergent graph partitions at runtime through handling partitions according to the topological order of SCCs in the DAG. Meanwhile, the vertices in each SCC are tried to be asynchronously handled along the directed paths, further reducing the number of rounds for each SCC to converge.

In the following, we first present some basic concepts and terminologies. Next, we will present the parallel execution model, followed by its implementation details. The performance analysis of the execution model is provided in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2017.2776115. Our execution model is able to get faster convergence speed and much lower I/O cost than existing solutions.

3.1 Basic Concepts

We first present the basic concepts used in our SCC-centric model.

Definition 1 (Update Dependency). *We call each edge in directed graph as an update dependency. In practice, for directed graph, if there is a directed edge $\langle v_i, v_j \rangle$, vertex v_i will send its state to vertex v_j and may cause the state update of v_j. Therefore, a vertex, e.g., v_j, becomes convergent only when all its precursors, e.g., v_i, are convergent.*

Definition 2 (SCC). *Let G=(V, E) denote a graph, where V=$\{v_l | 0 \leq l < n\}$ is a finite set of vertices, n is the number of vertices, and E=$\{\langle v_l, v_m \rangle | v_l \in V, v_m \in V\}$ is a set of edges. Let $w \rightarrow u$ denote that there is a path from w to u, i.e., $\forall t \in [0, k-1]$: $\langle v_t, v_{t+1} \rangle \in E$ and $v_0 = w$, $v_k = u$. Meanwhile, $w \nrightarrow u$ denotes that there is no path from w to u. A SCC, e.g., SCC_h, is a maximal subgraph of a directed graph G in which every pair of nodes is reachable from each other, i.e., $\forall v_i \in SCC_h \wedge v_j \in SCC_h : (v_i \rightarrow v_j \wedge v_j \rightarrow v_i)$ and $\forall v_i \in SCC_h \wedge v_k \notin SCC_h : (v_i \nrightarrow v_k \vee v_k \nrightarrow v_i)$.*

3.2 Execution Model

3.2.1 Overview

In a brief, our SCC-centric model first uses a DAG sketch to represent the graph G, where each vertex, or *SCC-vertex*, in DAG sketch represents a SCC. Specifically, the SCC-vertices and the edges among them form a DAG. The DAG sketch is G'=(V', E'), where V'=$\{SCC_h | SCC_h \subseteq G \wedge 0 \leq h < |S|\}$, $|S|$ is the number of SCCs in the graph G, E'=$\{\langle SCC_l, SCC_m \rangle | \exists v_l \in SCC_l \wedge v_m \in SCC_m : v_l \rightarrow v_m\}$.$SCC_h = \cup_{(x \in SCC_h \wedge y \in SCC_h)} Path(x, y)$, where $Path(x, y)$ is a path between two vertices x and y, i.e., $Path(x, y) = \langle v_0, v_1, \ldots, v_k \rangle$ is an ordered sequence of connected edges ensuring $v_0 = x, v_k = y$, $\forall t \in [0, k] : v_t \in SCC_h$. Note that any two paths have no intersecting edges.

For example, in Fig. 2, it has four SCCs. We can employ SCC-vertices to represent them and the directed graph is translated into a DAG sketch which contains four SCC-vertices. In DAG sketch, the directed edge $\langle SCC_i, SCC_j \rangle$ describes the update dependency between SCC_i and SCC_j. It means that the vertices in SCC_j can be processed only when the vertices in SCC_i are convergent. Otherwise, the vertices in SCC_i may send their new states to the vertices in SCC_j and trigger them to be reprocessed, although vertices in SCC_j may have been processed.

After that, in our model, vertices are handled with SCC as the processing unit and the SCCs are handled according to the topological order of the DAG sketch. In detail, each SCC is allowed to be processed only when all precursors of this SCC in the DAG sketch have been processed and become convergent. The vertices and edges in each SCC are organized as a set of paths without intersecting edges and the processing of each SCC takes the path, i.e., $Path(x, y)$, as the basic unit for processing. In other words, the vertices of each directed path, i.e., $Path(x, y)$, are stored and iteratively handled along their orders on the path, i.e., v_0, v_1, …, v_k, in

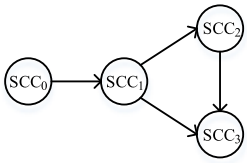


Fig. 3. An example to illustrate the update dependencies of SCCs.

an asynchronous way round by round until their states are convergent. When a vertex is processed, its new state is allowed to immediately work on its neighbors in the same round. After processing the last vertex, i.e., v_k, we may need to process them again along such an order at the next round until their states have been convergent.

In this way, when a SCC is arranged to be processed, its graph partitions will never receive new vertex state from partitions of the other SCCs, reducing the number of redundant vertex state updates. Besides, the vertices in each SCC also need fewer updates to absorb the states of the other vertices in the same SCC for efficient state propagation along directed paths, getting a faster convergence speed. Take Fig. 3 as an example, the processing order of the SCCs should be SCC_0, SCC_1, SCC_2 and SCC_3. Specifically, we first process SCC_0 until its vertex v_0 is convergent. Then vertices v_1, v_2 and v_3 in SCC_1 are processed along the directed path $v_1 \rightarrow v_2 \rightarrow v_3$ until their values are convergent. After processing SCC_1, we should process SCC_2 instead of SCC_3, because there is an update dependency from SCC_2 to SCC_3, which will cause the reprocessing of SCC_3 when vertices in SCC_2 are not convergent. In this way, each locally convergent SCC never needs to be reprocessed and vertices' states in each SCC also need fewer rounds for convergence.

In the following, we present the technical details on DAG sketch generation and parallel processing of SCCs.

3.2.2 Generation of DAG Sketch

The generation of DAG sketch is time consuming for large graph. Therefore, we propose a lightweight approach to generate DAG sketch in a parallel way.

It first divides the graph into N_w number of subgraphs for workers (threads in *thread pool*) to find local DAG sketch of each subgraph in a parallel way. In detail, as described in Algorithm 1, each worker first randomly takes a vertex as the root and traverses all the edges of its subgraph in a depth-first order (see Line 11), aiming to find the local SCCs and also generate local DAG sketch of each subgraph via contracting these local SCCs to SCC-vertices. Besides, each SCC is also divided into a set of paths, so as to allow efficient state propagation along the directed paths and also to improve the locality of vertex processing. In detail, each subgraph is visited according to the directed paths (see Lines 7 and 11). The visited edges and related vertices of each traverse are gathered one by one to form a path (see Line 3). Note that the SCCs are stored via using the same-sized partitions in terms of number of edges (see Line 19). The giant SCC is evenly stored into several partitions for parallel processing and the highly-connected small SCCs (may contain only one vertex) are stored together for balanced load and low communication cost. This step only needs to access each vertex and edge for exactly once, and its time complexity is O (V+E), where V and E are the number of vertices and edges, respectively.

Algorithm 1. DAG Sketch Generation Algorithm

```
1:  procedure PARTITION(G, V_i, Path, acc, Partition P) /*Path is a stack*/
2:    Set V_i as visited.
3:    Push(Path, V_i) /*Push V_i into Path*/
4:    High[V_i] ← acc
5:    Low[V_i] ← acc
6:    acc ← acc + 1
7:    V^suc ← FindSuccessors(V_i)
8:    for V_j^suc ∈ V^suc and ⟨V_i, V_j^suc⟩ is unvisited do
9:      Set edge ⟨V_i, V_j^suc⟩ as visited.
10:     if V_j^suc is unvisited then
11:       Partition(G, V_j^suc, Path, acc, P)
12:       Low[V_i] ← Min(Low[V_j^suc], Low[V_i])
13:     else if V_j^suc ∈ Path then
14:       Low[V_i] ← Min(High[V_j^suc], Low[V_i])
15:     end if
16:   end for
17:   if Low[V_i] = High[V_i] then /*Vertices from V_i to the top of Path are in a SCC*/
18:     SCC ← Pop(Path, V_i) /*Pop all vertices from Path until V_i* is met*/
19:     Insert(P, SCC) /*Create a new partition when P is full*/
20:   end if
21: end procedure
```

After that, all local DAG sketches are combined into the global DAG sketch via combining some local SCCs to the global ones. For this goal, we take each local SCC as a node and also use the above algorithm to further translate the graph consisting of local DAG sketches into the global DAG sketch. Note that this intermediate graph is much smaller than the original graph and is contracted using a worker. Therefore, the generation of DAG sketch only needs to access the graph twice. This overhead is often a small proportion of the total execution time because real-world graph is usually needed to be processed by an iterative algorithm for several rounds, even hundreds of rounds for convergence.

3.2.3 Parallel Processing of SCCs

After generating DAG sketch, it begins to process SCCs in a parallel way, because modern machines have multi-core capabilities. Thus, we propose to efficiently parallelize our model at SCC level and within SCC.

SCC-Level Parallel Processing. According to our SCC-centric model, all SCCs should be handled according to the topological sequence of the DAG sketch. In order to parallelize the execution of SCCs for our SCC-centric model, we propose an approach to handle the SCCs based on levels, where there is no update dependency between the SCCs at the same level. In this way, all SCCs can be easily handled in a parallel way level by level, where the SCCs at the same level can be assigned to workers for parallel processing.

In detail, it first divides the DAG sketch into levels as described in Algorithm 2. This process is repeatedly done until all SCCs are assigned with a level number (see Lines 2-7), and the SCCs at each level only depend on the

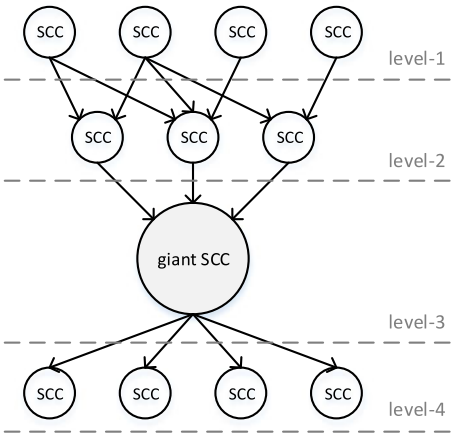


Fig. 4. Processing of a giant SCC in a single level.

SCCs at the lower level. Fig. 4 gives an example to illustrate it. The SCCs without any precursors are at the level 1. Then the remaining SCCs without precursors are at the level 2, etc. The SCC with the smallest level number is assigned with the highest priority for processing via making it at the front of the task queue (see Line 5).

Algorithm 2. Level Generation Algorithm

```
1: procedure FindLevels(DAG)
2:   while IsNotEmpty(DAG) do /*Not empty*/
3:     Lev ← GetAllSCCHasNoInDegree(DAG)
4:     Ascending(Lev)
5:     Enqueue(Levels, Lev)
6:     DeleteAllSCCInLevel(DAG, Lev)
7:   end while
8:   OutputLevels(Levels)
9: end procedure
```

One thing is worth further discussions. For real-world graph [17], the DAG sketch may also have skewed degree distributions, where some SCC-vertices, or called *hub SCC-vertices*, have much more out-edges than others. It means that many SCC-vertices are dependent on hub SCC-vertices. The delayed processing of hub SCC-vertices may induce many workers to be idle. In order to increase the parallelism, the SCCs in each level are arranged in ascending order according to the number of their out-edges in DAG sketch (see Line 4). In this way, the SCCs with the most out-edges at each level can be first processed. Their successors in the next level can be processed when some workers become idle.

After that, as described in Algorithm 3, it begins to process SCCs at each level in a parallel way until all SCCs have been processed (see Line 2). Once there are idle workers, the SCCs are taken out from the task queue (see Line 3) for processing one by one in order. In order to spare redundant overhead, only the non-convergent SCCs are loaded into memory for processing (see Line 4). Meanwhile, a SCC is tried to be allocated to the free worker with the most number of its precursors (see Line 8). The results of each SCC for its successors are tried to be buffered in memory. In this way, when its successors are loaded into memory, the processing of them needs less I/O cost to access the results. When the memory is not enough to buffer all results, some ones need to be written back to disk. In order to maximize its performance, the buffered results of a SCC are swapped out of memory when it has the least number of non-convergent successors.

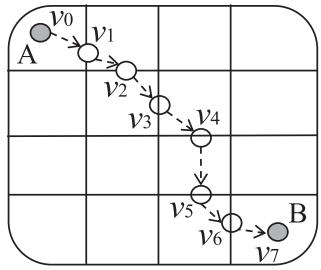


Fig. 5. Inefficient vertex state propagation along directed paths within giant SCC (Assume a giant SCC is divided into several partitions and vertex v_0 has aggregated a certain number of vertices' states in partition A and needs to propagate them to the vertices in partition B for convergence).

Algorithm 3. Level-Based Graph Processing Method

```
1: procedure ProcessLevels(Levels)
2:   while Levels is not empty do
3:     SCC ← Dequeue(Levels)
4:     if SCC is not convergent then
5:       if |SCC| > α · S_M then
6:         /*Give all free workers for SCC*/
7:       else
8:         /*Allocate a free worker for SCC*/
9:       end if
10:    end if
11:  end while
12: end procedure
```

In some directed graphs, some SCCs are much larger than others and the size of giant SCC may be larger than the size of memory, i.e., S_M. It not only induces a large amount of out-of-memory data access when repeatedly processing its vertices in an iterative way, but also may swap small SCCs out of memory or cache, inducing poor performance. Consequently, giant SCC needs special treatment. The size of each SCC is evaluated at the preprocessing stage. A SCC is identified as a giant one, if its size, i.e., $|SCC|$, is larger than $\alpha \cdot S_M$ ($\alpha \geq 1/2$) (see Line 5). After that, we try to employ a level to process a single giant SCC as Fig. 4. All remaining memory and free workers are incrementally assigned to a single giant SCC (see Line 6). In detail, when the processing of a SCC is ended, its memory and CPU resource will also be incrementally allocated to the giant SCC. In this way, the processing of giant SCC not only has less opportunity to compete memory and cache with small SCCs, but also can be finished in a faster way. Then it has lower memory/cache miss rate and faster convergence speed.

Inner-SCC Parallel Processing. As the above described, the giant SCC contains much more vertices than other SCCs and needs to be divided into several partitions for parallel processing. However, with existing parallel solutions, as described in Fig. 5, the vertices on the directed path from v_0 to v_7 may be divided into seven partitions and processed in a random order, e.g., v_7, v_6, ..., v_1, v_0, as the vertices of different partitions. As a result, the vertices' states still cannot be efficiently propagated along directed paths in giant SCC although with the asynchronous processing way. It needs

seven rounds to process the whole SCC for the states accumulated by v_0 to reach v_7, because the states of already-processed vertices can only be updated in the next round.

In practice, the real-world graphs [17] often have the power-law property regarding node degrees. It means that a small portion of hub vertices have much higher degrees than the others and the states of most vertices will pass through them to reach each other for convergence. Therefore, in order to reduce the reprocessing cost of giant SCC, we propose a processing way to quickly propagate the vertex states passing through the directed paths between hub vertices.

It first needs to build a backbone structure that consists of the hub vertices and paths between them, which is extracted from the giant SCC and spans all its partitions. In essence, this structure serves as a fast track for cross-partition state propagation. To get this goal, it identifies the hub vertices (its degree is larger than a user given threshold) from each partition of the giant SCC, then repartitions this SCC. Its hub vertices and the edges on the paths between them are divided into the same partition, and the remaining edges of this SCC are divided into other same-sized partitions. After that, it asynchronously processes each partition, as described in Section 3.2.1, via handling vertices along the directed paths as much as possible within each round. In this way, most vertex state propagations that may otherwise take place in a random order by traditional asynchronous methods can now be done within the same partition (backbone structure) along the directed paths between hub vertices, accelerating the convergence of giant SCC.

3.2.4 Supporting for Evolving Graph

In the real world, the graph structure may have changes. In order to efficiently support evolving graph, we first employ a lightweight incremental approach to maintain the DAG sketch. If the graph has structure changes, it only needs to identify the SCCs with changes and takes these SCCs as a new subgraph to repartition them and also reconstruct the DAG sketch of this subgraph via the algorithm described in Section 3.2.2. The SCCs with structure changes are marked and need reprocessing. In this way, it allows to incrementally maintain the DAG sketch according to the removed or added edges so as to efficiently support evolving graph. Note that it only needs to get DAG sketch for exactly once for static graph at the preprocessing stage. After that, it can be reused for different graph algorithms.

Meanwhile, in our model, when a SCC has no update for its successors in the DAG sketch to handle, these successive SCCs will not be loaded into memory for processing. Then it only needs to process the SCCs with structure changes and the SCCs dependent on the changed SCCs. Note that the other SCCs are still convergent and do not need to be processed. Therefore, our model is able to efficiently support the analysis of evolving graph as well.

3.3 Implementation of DGraph

Our DGraph system is implemented by modifying a state-of-the-art graph processing system, i.e., PathGraph [13], [20], and incorporating our proposed design.

DGraph is executed as described in Algorithm 4. It first divides the graph into several partitions and generates DAG sketch via Algorithm 1 (see Line 3). To effectively store the DAG sketch and vertices of each SCC, a key-value table is established. Each table entry represents a SCC indexed by its key and with three other fields to describe corresponding information. The first two fields indicate the keys of its precursors and successors, respectively. The third field stores the keys of its partitions. The information of each partition is also stored in a key-value table and its each data item indicates a vertex and contains four fields: vertex ID, SCC ID, state value, and edges assigned in this partition including associated edge information, e.g., priority.

Algorithm 4. Execution of DGraph

```
1:  procedure EXECUTOR(G)
2:    if DAG is not generated for G then
3:      Divide graph G and generate DAG
4:    end if
5:    if G has structure changes then
6:      SCC_c ← Identify all SCCs with changes
7:      Divide subgraph SCC_c and generate DAG_c
8:      Combine local DAG_c into DAG
9:    end if
10:   Levels ← FindLevels(DAG)
11:   ProcessLevels(Levels)
12: end procedure
```

The graph division and DAG sketch generation is done for exactly once for static graph (see Line 2). If there are structure changes, e.g., a edge $< v_i, v_j >$ is added/deleted, it needs to identify the SCCs with changes according to the added/deleted edges, e.g., the SCC with vertex v_i or v_j is added into a set SCC_c (see Line 6). Then it incrementally repartitions the subgraph SCC_c and regenerates a local DAG sketch for SCC_c using Algorithm 1 (see Line 7). Finally, the DAG sketch is maintained via combining local DAG sketch, i.e., DAG_c, with the other DAG sketches without changes (see Line 8). After that, it begins to process the graph algorithm as needed. At this moment, the partitions of SCCs are loaded into memory for processing according to the DAG sketch (see Line 11).

4 Experimental Evaluation

In this section, we present experimental evaluation of DGraph in comparison with state-of-the-art schemes for static graph and evolving graph. Also, our approach significantly outperforms the existing SCC-based approaches [39], [40], [43]. In Appendix B, available in the online supplemental material, we also provide some additional experiments in comparison with those SCC-based solutions [39], [40], [43].

4.1 Experimental Setup

The hardware platform used in our experiments is a server containing 2-way 8-core 2.60 GHz Intel Xeon CPU E5-2670, running a Linux operation system with kernel version 2.6.32. Its memory is 64 GB and the secondary storage for it is disk. It spawns a worker for each core to run benchmarks. The program is compiled with cmake version 2.6.4 and gcc version 4.8.1. In experiments, four common graph algorithms are employed as benchmarks: (1) PageRank [3], a popular algorithm for ranking web pages; (2) Adsorption [4], a graph-based label propagation algorithm to

TABLE 2
Data Sets Properties

Data sets	Vertices	Edges	Raw size	Number of SCCs	Vertex(Edge) ratio of the largest SCC
soc-pokec [44]	1,632,803	30,622,564	213 MB	325,893	79.9%(95.3%)
soc-LiveJour [44]	4,847,571	68,993,773	527 MB	971,232	79.0%(95.4%)
enwiki [45]	4,206,757	101,355,853	772 MB	449,738	89.0%(97.8%)
webbase [45]	118,142,121	1,019,903,024	9.6 GB	41,126,852	45.6%(61.8%)
uk2007 [45]	105,896,268	3,738,733,633	32.1 GB	21,398,038	64.8%(75.3%)
uk-union [45]	133,633,040	5,507,679,822	48.5 GB	22,753,644	65.3%(78.9%)
yahoo-web [46]	1,413,511,394	6,636,600,779	66.9 GB	1,329,477,647	3.5%(15.7%)
clueweb12 [45]	978,408,098	42,574,107,469	318.0 GB	135,223,661	79.2%(89.6%)
hyperlink14 [47]	1,727,629,502	64,422,273,418	480.0 GB	206,147,524	19.0%(30.2%)
hyperlink12 [47]	3,563,415,266	128,736,182,575	957.2 GB	223,305,262	51.3%(67.4%)

provide personalized recommendation; (3) SSSP [5], which finds single source shortest paths in a graph; (4) K-Core [6], which is often used for community detection and so on.

The characteristic statistics (i.e., the number of SCCs, the number of vertices in the largest SCC, and the number of edges in the largest SCC) about the real-world graphs of the experiments are specified in Table 2. In these datasets, the ratio of the largest SCC's vertices (edges) against the total number of vertices (edges) ranges from 3.5 percent (15.7 percent) to 89.0 percent(97.8 percent). Note that the number of SCCs in soc-pokec, soc-LiveJour and clueweb12 is increased from 325,893 to 135,223,661, although the ratios of the largest SCC's vertices for them are almost the same. The first six graphs can be totally loaded into memory by DGraph and PathGraph. The other data sets cannot be totally loaded into memory and are used to evaluate the performance of DGraph and PathGraph for out-of-core computing (see Section 4.3).

To understand the performance of our SCC-centric execution model, two other versions of DGraph are realized. *DGraph-s* uses the traditional synchronous way [20] to process each partition. *DGraph-t* employs the traditional asynchronous way [11] instead of our proposed asynchronous way. Additionally, we compare the following two processing ways built in PathGraph [20]: 1) *PathGraph-s*, which employs the synchronous way [20] to process each partition; 2) *PathGraph-a*, which uses the traditional asynchronous way [11]. Besides, the performance of DGraph is further compared with ODS [21], GridGraph [11] and NXgraph [12], which are three state-of-the-art graph processing systems on single PC. ODS is based on GraphChi and uses the vertex-centric model. GridGraph and NXgraph employ the edge-centric model. Note that both ODS and GridGraph in the experiments use the asynchronous graph processing way.

4.2 Convergence Speed

The execution time for different solutions to converge is described in Table 3. We can find that our approach achieves performance improvements of 1.46-8.37 times. DGraph has the best performance. For K-Core over webbase, the execution time needed by DGraph is only 11.9, 47.9 and 14.8 percent of PathGraph-s, DGraph-s and PathGraph-a, respectively. In reality, the shortest execution time of DGraph mainly benefits from the smallest number of updates to converge.

Table 4 depicts the number of vertex state updates for convergence with the current solutions. As shown in the table, DGraph needs much smaller number of updates for convergence than existing solutions and can reduce the number of updates by 33.2-91.0 percent. This is because,

TABLE 3
Execution Time of Different Benchmarks with the Current Solutions (Time in Seconds)

Data sets		soc-pokec	soc-LiveJour	enwiki	webbase	uk2007	uk-union
PageRank	PathGraph-s	3.305	6.375	7.846	203.511	309.710	355.256
	DGraph-s	2.164	4.139	5.374	88.024	142.583	170.623
	PathGraph-a	1.865	5.459	4.947	161.588	273.305	315.481
	DGraph	0.791	2.390	1.962	62.510	108.135	127.224
Adsorption	PathGraph-s	7.196	13.792	20.221	393.055	513.020	1,241.515
	DGraph-s	4.723	8.858	13.756	143.085	250.627	622.403
	PathGraph-a	4.885	10.016	14.290	280.248	425.364	1,123.278
	DGraph	1.750	4.104	4.763	78.721	163.839	446.318
SSSP	PathGraph-s	3.091	5.241	7.857	247.667	417.262	1,113.644
	DGraph-s	1.994	3.331	5.275	86.870	190.443	554.039
	PathGraph-a	1.908	3.970	4.566	170.059	293.804	990.631
	DGraph	0.715	1.514	1.941	50.046	110.650	411.515
K-Core	PathGraph-s	5.528	12.224	14.795	638.827	2,052.624	5,176.281
	DGraph-s	3.625	7.902	9.778	159.413	918.506	2,831.379
	PathGraph-a	3.848	10.528	12.105	515.858	1,708.479	4,181.902
	DGraph	1.913	4.386	4.732	76.323	632.376	1,502.065

TABLE 4
Number of Updates for Convergence with Different Solutions

Data sets		soc-pokec	soc-LiveJour	enwiki	webbase	uk2007	uk-union
PageRank	PathGraph-s	217,979,334	465,366,816	499,348,594	15,358,475,730	22,714,749,486	26,305,663,924
	DGraph-s	131,749,603	270,572,102	333,564,860	5,193,729,941	9,735,947,983	12,069,743,659
	PathGraph-a	130,214,270	368,415,396	298,222,077	12,814,212,100	16,453,401,116	18,762,078,816
	DGraph	43,739,799	133,969,235	109,851,540	3,836,319,380	5,121,628,808	6,340,361,542
Adsorption	PathGraph-s	255,370,545	560,379,207	828,548,778	18,868,712,477	24,411,207,698	53,780,555,200
	DGraph-s	150,257,967	322,067,015	491,125,011	6,281,322,982	9,525,867,192	25,373,988,982
	PathGraph-a	162,235,405	395,561,793	501,935,691	12,756,022,461	17,103,010,096	39,992,571,278
	DGraph	49,087,020	114,010,451	138,786,892	3,220,375,340	5,075,339,215	14,724,148,698
SSSP	PathGraph-s	241,692,761	442,098,475	618,721,490	25,020,246,835	37,131,851,656	97,174,460,608
	DGraph-s	150,519,047	258,369,929	397,700,931	8,549,788,051	15,411,271,318	41,692,584,869
	PathGraph-a	137,978,215	297,834,762	341,534,261	15,373,398,854	24,602,699,255	73,538,606,135
	DGraph	46,690,056	98,431,546	127,277,993	4,324,883,939	8,363,100,447	25,696,639,765
K-Core	PathGraph-s	195,936,480	479,492,910	556,566,175	25,909,748,556	80,830,727,260	205,770,293,120
	DGraph-s	116,461,720	301,076,586	348,564,939	5,946,643,953	33,957,120,881	102,535,130,654
	PathGraph-a	163,907,072	401,071,845	458,561,031	19,650,885,738	52,480,636,301	157,150,466,110
	DGraph	71,363,944	157,602,301	168,225,368	2,337,403,335	15,873,984,897	49,860,389,416

both PathGraph-s and PathGraph-a still need many iterations to be convergent although only a small number of vertices are not convergent. Meanwhile, we can observe that the asynchronous way needs fewer updates than the synchronous way. For example, the number of updates needed by DGraph is only 28.3 percent of DGraph-s for Adsorption over enwiki for faster vertex state propagation.

We further make the following observations.

First, our execution model is sensitive to the structure of directed graph. Take PageRank as an example, the number of updates required by DGraph-s is only 33.8 percent of PathGraph-s over webbase, while the ratio is 66.8 percent over enwiki. Now, we discuss its reasons. Assume two opposite extreme scenarios. If the whole graph is a giant SCC, i.e., with all the vertices and edges of this graph, this method degenerates to the traditional model and its efficiency is the same as the traditional model. While, if all SCCs in the graph contain only one vertex, the graph is a DAG. It means that there is no loop in the graph and DGraph-s only needs to process the graph for one round. Therefore, DGraph-s is more feasible for the graph structure which contains many small SCCs.

Second, the results show the effectiveness of our SCC-centric model for most of directed graphs in the real world. It is because that the graph in the real world often consists of giant SCCs and small SCCs. Specifically, the number of different sized SCCs has skewed distribution [17]. Some SCCs are very large, where the vertices of a SCC even occupy most of the whole graph's vertices. The number of giant SCCs is often very small. While, others SCCs are small, consisting of two or even one vertex, but the number of them is very large.

In order to demonstrate the superiority of our asynchronous way against the traditional asynchronous way, the performance of DGraph and DGraph-t is also compared. The relative execution time for DGraph and DGraph-t is depicted in Fig. 6. We can find that DGraph reduces the execution time of DGraph-t even up to 57.5 percent for Adsorption over enwiki.

4.3 Performance for Out-of-Core Computing

Fig. 7 gives the execution time of different graph algorithms with PathGraph-s, DGraph-s, PathGraph-a and DGraph over yahoo-web, clueweb12, hyperlink14 and hyperlink12, respectively. In this way, it is able to evaluate the performance of different solutions when the whole graph cannot be loaded into memory. As shown in the figure, our approach is able to significantly reduce the execution time. Take yahoo-web as an example, the execution time of PageRank with PathGraph-s is spared 64.1, 27.4 and 86.4 percent by DGraph-s,

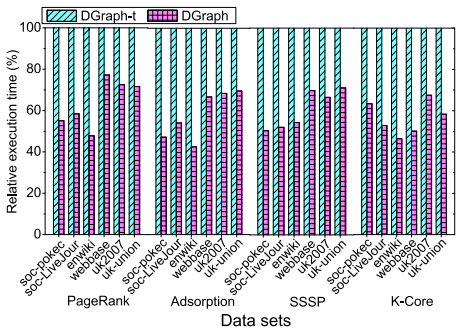


Fig. 6. Relative execution time of different graph algorithms using DGraph against DGraph-t

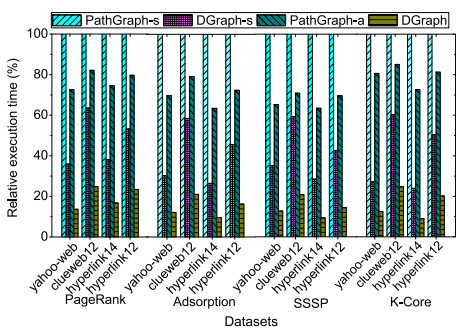


Fig. 7. Relative execution time of out-of-core graph algorithms using different solutions.

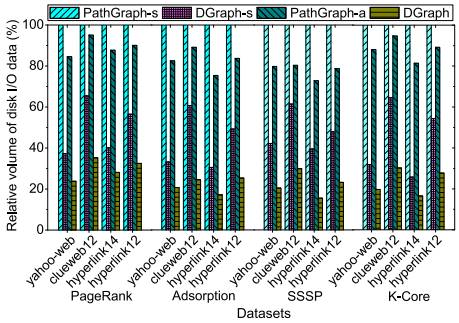


Fig. 8. Relative I/O overhead of out-of-core graph algorithms using different solutions.

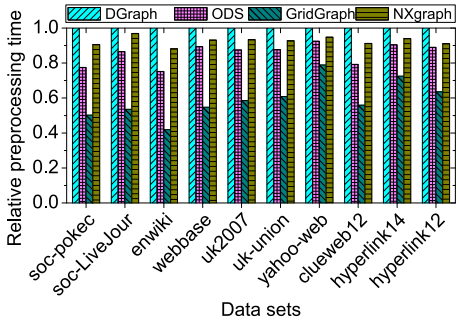


Fig. 9. Preprocessing time for different data sets.

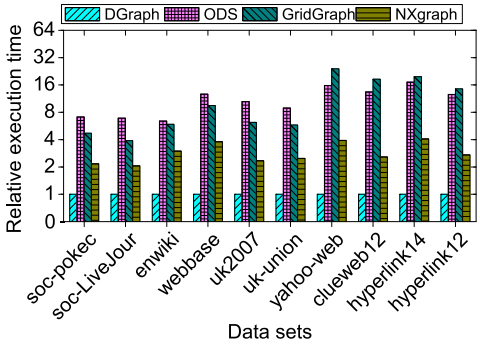
PathGraph-a, DGraph, respectively. We observe that the shortest execution time of DGraph mainly comes from two reasons. First, it needs the least number of updates for convergence as described in Table 4. Second, DGraph also needs the lowest I/O cost in comparison with them.

The I/O cost of different solutions over yahoo-web, clueweb12, hyperlink14 and hyperlink12 is depicted in Fig. 8. We can observe that DGraph-s is able to reduce the I/O cost of PageRank up to 62.7 percent against that of PathGraph-s for yahoo-web, thanks to less redundant processing of locally convergent vertices and better locality. Besides, for PageRank over yahoo-web, the I/O cost of DGraph-s is also reduced 36.2 percent by DGraph because of smaller I/O cost for giant SCC to converge. Thus our approach is very suitable to the out-of-core computing.

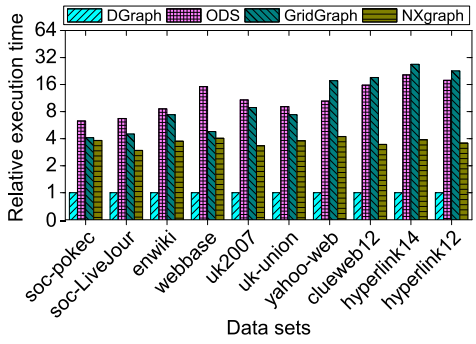
4.4 Comparison with Other Systems

We first evaluate the preprocessing time of different systems. This metric consists of the time to load data into the memory, and build and partition the graph. As shown in Fig. 9, DGraph takes more preprocessing time than the other systems, because DGraph needs to construct DAG sketch via contracting SCCs when partitioning the graph. However, DAG sketch brings significant benefits such as lower data access cost and higher convergence speed. It is because that the generation of DAG sketch only needs to traverse the graph twice and can be proceeded in a parallel way, while ODS, GridGraph and NXgraph have to traverse the graph much more than twice for convergence at runtime (see Appendix A.1, available in the online supplemental material). Besides, the DAG sketch can be reused in DGraph after preprocessing when the graph is static.

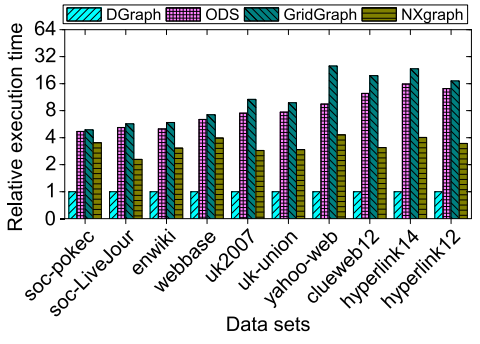
Fig. 10 gives the execution time of different graph algorithms, where all data sets are tried to be loaded into memory in advance. The computation is finished only when all vertices have been convergent. We can find that DGraph always gets the least execution time for each graph algorithm. DGraph offers dramatic improvements of 4.7-20.5, 3.9-26.9 and 2.06-4.31 times in comparison with ODS, GridGraph and NXgraph, respectively. It is because that ODS, GridGraph



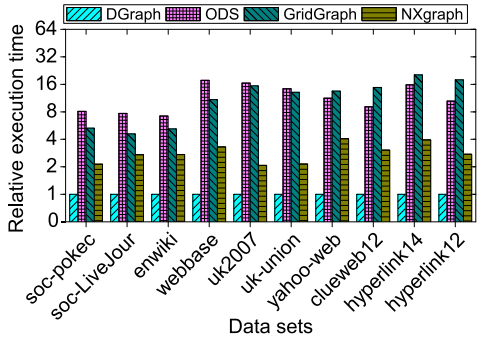
(a) PageRank



(b) Adsorption



(c) SSSP



(d) K-Core

Fig. 10. Relative execution time of graph processing algorithms with different solutions.

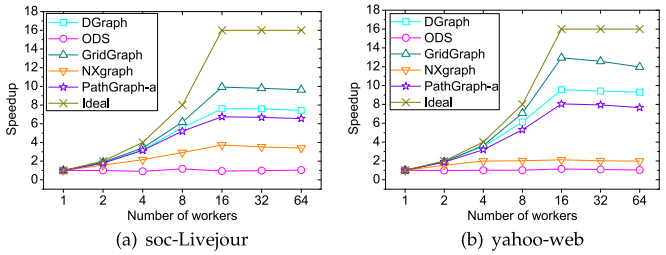


Fig. 11. Scalability of PageRank with different solutions.

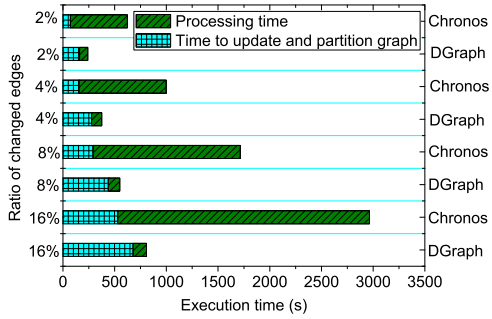


Fig. 12. Execution time of PageRank when different ratios of uk-union's edges are removed/added.

and NXgraph all need much more updates for convergence and higher volume of data to access for the repeated graph partition processing. Besides, we can find that NXgraph needs less time to converge than ODS and GridGraph. Take Adsorption as an example, the execution time of NXgraph is only 19.0 and 14.5 percent of ODS and GridGraph over hyperlink14, respectively, for less data access time. However, NXgraph suffers from high synchronization cost. It incurs slow vertex state propagation in NXgraph. As a result, for the graph with a high diameter, NXgraph needs to execute a very large number of iterations, each of which requires to stream the entire edge list for handling.

Fig. 11 shows the scalability of different systems. As depicted in the figure, DGraph almost has the same scalability as GridGraph, much better than the other systems. Besides, we can observe that NXgraph has a better scalability on soc-Livejour than yahoo-web. It is because that NXgraph can exploit the parallelism of multicore when the small graph, i.e., soc-Livejour, can be completely loaded into memory. However, for yahoo-web, the scalability of NXgraph is limited by disk I/O. For both soc-Livejour and yahoo-web, ODS may need more time for convergence as we increase the number of workers. According to the profiled runtime information, ODS nearly saturates the bandwidth for main memory and magnetic disk with only one core, incurring poor scalability.

4.5 Performance for Evolving Graph

The performance of DGraph is also evaluated for the analysis of a growing directed graph against Chronos [18], [19], which is a state-of-the-art system supports evolving graph analysis. In the experiments, we first let both DGraph and Chronos process PageRank over uk-union for once. Then we randomly add or remove some edges over uk-union and evaluate the execution time for different solutions to converge again.

Fig. 12 depicts the execution time of PageRank with DGraph and Chronos when different ratios of edges have been randomly removed or added over uk-union. As shown in the figure, when the change ratio of the whole graph's edges is smaller, the execution time of both DGraph and Chronos decrease, because they can incrementally handle the graph updates. Besides, we can observe that the overhead time of DGraph is longer than Chronos because DGraph needs additional time to update DAG sketch according to the graph structure changes. However, Chronos still needs much more execution time than DGraph because of much more vertex state updates to handle and higher I/O cost for convergence. For example, when the edge update ratio is 2 percent, the overhead time of Chronos is 72.8 seconds less than 160.0 seconds needed by DGraph, yet DGraph only needs 80.4 seconds to process graph vertices much less than 550.5 seconds of Chronos.

5 Conclusion

Iterative directed graph processing has been an important and fruitful research topic. However, the current vertex/edge-centric execution models suffer from highly redundant disk I/O accesses and slow convergence speed. This paper takes advantage of an interesting structure (SCC) in a directed graph, and proposes a SCC-centric model to improve the efficiency of iterative directed graph processing. With SCC, the unique feature of our approach is that the vertices never need to be processed again when they become locally convergent within each SCC, which significantly eliminates the redundant disk I/O accesses and improves convergence speed. Experimental results show that our model outperforms the state-of-the-art solutions for iterative directed graph algorithms.

In the future, we have several optimizations to be done. First, recent advances in accelerators (e.g., GPU and FPGA) and hybrid memory (e.g., DRAM+NVM) for parallel graph processing [48] motivate us to extend our approach to heterogeneous platforms, so as to efficiently handle large-scale directed graphs on a single machine. We are recently extending our approach to a heterogeneous platform consisting of GPUs/FPGAs for efficient directed graph analysis. Second, we will also extend our approach to distributed platform for high scalability, because the sizes of many real-world directed graphs are up to hundreds of terabytes. For efficient communication, RDMA will be used and efficient dedicated communication scheme will also be developed for it. Finally, we will research how to further optimize our approach for evolving directed graph analysis.

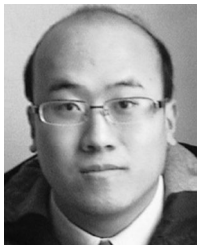
Acknowledgments

This paper is supported by National Natural Science Foundation of China under grant No. 61702202, 61628204 and 61732010, China Postdoctoral Science Foundation Funded Project under grant No. 2017M610477 and 2017T100555.

References

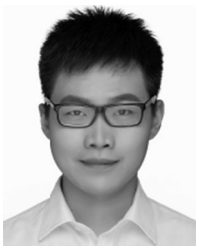
[1] H. Chen, H. Jin, and X. Cui, "Hybrid followee recommendation in microblogging systems," *Sci. China Inf. Sci.*, vol. 60, no. 012102, pp. 1–14, 2017.

[2] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. A. Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proc. 22nd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2017, pp. 389–404.
[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Technical report, Stanford Digital Library Technologies Project, 1998.
[4] S. Baluja, et al., "Video suggestion and discovery for YouTube: Taking random walks through the view graph," in *Proc. 17th Int. Conf. World Wide Web*, 2008, pp. 895–904.
[5] U. Meyer, "Single-source shortest-paths on arbitrary directed graphs in linear average-case time," in *Proc. 12th Annu. ACM-SIAM Symp. Discr. Algorithms*, 2001, pp. 797–806.
[6] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," *Proc. VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.
[7] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[8] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 472–488.
[9] W. S. Han, et al., "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 77–85.
[10] J. Cheng, Q. Liu, and Z. Li, "VENUS: Vertex-centric streamlined graph computation on a single PC," in *Proc. IEEE Int. Conf. Data Eng.*, 2015, pp. 124–134.
[11] X. Zhu, W. Han, and W. Chen, "GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.
[12] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 409–420.
[13] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee, "Fast iterative graph computation: A path centric approach," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 401–412.
[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, and N. Leiser, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[15] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.
[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," in *Proc. 26th Conf. Uncertainty Artif. Intell.*, 2010, pp. 1–10.
[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[18] W. Han, et al., "Chronos: A graph engine for temporal graph analysis," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.
[19] Y. Miao, et al., "ImmortalGraph: A system for storage and analysis of temporal graphs," *ACM Trans. Storage*, vol. 11, no. 3, pp. 1–14, 2015.
[20] P. Yuan, C. Xie, L. Liu, and H. Jin, "PathGraph: A path centric graph processing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2998–3012, Oct. 2016.
[21] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 507–522.
[22] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 281–292, 2014.
[23] M. Han, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.
[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[25] Y. Zhang, X. Liao, H. Jin, and G. Tan, "SAE: Toward efficient cloud data analysis service for large-scale social networks," *IEEE Trans. Cloud Comput.*, vol. 5, no. 3, pp. 563–575, Jul.–Sep. 2017.
[26] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 410–424.
[27] Y. Zhang, X. Liao, H. Jin, L. Gu, G. Tan, and B. B. Zhou, "HotGraph: Efficient asynchronous processing for real-world graphs," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 799–809, May 2017.
[28] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenters," in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 1397–1407.
[29] Y. Zhang, X. Liao, H. Jin, and G. Min, "Resisting skew-accumulation for time-stepped applications in the cloud via exploiting parallelism," *IEEE Trans. Cloud Comput.*, vol. 3, no. 1, pp. 54–65, Jan.–Mar. 2015.
[30] Y. Zhang, X. Liao, H. Jin, G. Tan, and G. Min, "Inc-part: Incremental partitioning for load balancing in large-scale behavioral simulations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1900–1909, Jul. 2015.
[31] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to fuse for distributed graph-parallel computation," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 194–204.
[32] R. Cheng, et al., "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2012, pp. 85–98.
[33] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 2014–2025, 2013.
[34] L. Wang, F. Yang, L. Zhuang, H. Cui, F. Lv, and X. Feng, "Articulation points guided redundancy elimination for betweenness centrality," in *Proc. 21st ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, pp. 1–7.
[35] R. Puzis, Y. Elovici, P. Zilberman, S. Dolev, and U. Brandes, "Topology manipulations for speeding betweenness centrality computation," *J. Complex Netw.*, vol. 3, no. 1, pp. 84–112, 2015.
[36] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: Scalable reachability index for large graphs," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 276–284, 2010.
[37] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang, "DAG reduction: Fast answering reachability queries," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 375–390.
[38] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph homomorphism revisited for graph matching," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 1161–1172, 2010.
[39] M. Brinkmeier, "Distributed calculation of pagerank using strongly connected components," in *Proc. Int. Workshop Innovative Internet Community Syst.*, 2005, pp. 29–40.
[40] V. Bloemen, A. Laarman, and J. van de Pol, "Multi-core on-the-fly SCC decomposition," in *Proc. 21st ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, pp. 1–8.
[41] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (SCC) in small-world graphs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–11.
[42] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 8, pp. 2091–2100, Aug. 2014.
[43] M. Brinkmeier, "Pagerank revisited," *ACM Trans. Internet Technol.*, vol. 6, no. 3, pp. 282–301, 2006.
[44] Stanford large network dataset collection. 2016. [Online]. Available: http://snap.stanford.edu/data/index.html
[45] Laboratory for web algorithmics. datasets. 2016. [Online]. Available: http://law.di.unimi.it/datasets.php
[46] Yahoo! Lab. datasets. 2016. [Online]. Available: http://webscope.sandbox.yahoo.com/catalog.php?datatype=g
[47] Web data commons—hyperlink graphs. 2016. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/
[48] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.



Yu Zhang received the PhD degree in computer science from Huazhong University of Science and Technology (HUST), in 2016. He is now a postdoctor in School of Computer Science, HUST. His research interests include big data processing, cloud computing, and distributed systems. His current topic mainly focuses on application-driven big data processing and optimizations.

Xiaofei Liao received the PhD degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2005. He is now a professor in the School of Computer Science and Engineering, HUST. His research interests include the areas of system virtualization, system software, and Cloud computing. He is a member of the IEEE.



Xiang Shi received the BS degree from the School of Computer, Wuhan University, China, in 2014. He is working toward the master's degree in computer science from Huazhong University of Science and Technology (HUST). His current research interests include big data and graph processing.

Hai Jin received the PhD degree in computer engineering from Huazhong University of Science and Technology, in 1994. He is a Cheung Kung Scholars chair professor of Computer Science and Engineering, Huazhong University of Science and Technology (HUST) in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with the University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of CCF, senior member of the IEEE, and a member of the ACM. He has co-authored 22 books and published more than 800 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Bingsheng He received the bachelor degree from Shanghai Jiao Tong University, in 2003 and the PhD degree from Hong Kong University of Science and Technology, in 2008. He is currently an associate professor at the Department of Computer Science, National University of Singapore. He has served as a PC member for international conferences in databases (e.g., ACM SIGMOD, VLDB, IEEE ICDE), cloud computing (e.g., ACM SoCC) and parallel and distributed systems (e.g., SC, HPDC and IPDPS), and as a demo co-chair in VLDB 2017, PC co-chair in IEEE CloudCom 2014/2015 and HardBD 2016. His current research interests include big data management systems (with special interests in cloud computing and emerging hardware systems), parallel and distributed systems, and cloud computing.