DiGraph: An Efficient Path-based Iterative Directed Graph Processing System on Multiple GPUs

Yu Zhang
Huazhong University of Science and Technology, China*
zhyu@hust.edu.cn

Xiaofei Liao
Huazhong University of Science and Technology, China*
xfliao@hust.edu.cn

Hai Jin
Huazhong University of Science and Technology, China*
hjin@hust.edu.cn

Bingsheng He
National University of Singapore, Singapore
hebs@comp.nus.edu.sg

Haikun Liu
Huazhong University of Science and Technology, China*
hkliu@hust.edu.cn

Lin Gu
Huazhong University of Science and Technology, China*
lingu@hust.edu.cn

Abstract

Many systems are recently proposed for large-scale iterative graph analytics on a single machine with GPU accelerators. Despite of many research efforts, for iterative directed graph processing over GPUs, existing solutions suffer from slow convergence speed and high data access cost, because many vertices are ineffectively reprocessed for lots of rounds so as to update their states according to other active vertices regardless of their dependencies. In this paper, we propose a novel and efficient iterative directed graph processing system on a machine with the support of multiple GPUs. Compared with existing systems, the unique feature of our system is that it takes advantage of the dependencies between vertices in three novel ways. First, it represents a directed graph into a set of disjoint hot/cold directed paths and takes the path as the basic parallel processing unit, so as to help efficient vertex state propagation along the paths over GPUs for faster convergence speed and higher utilization ratio of the loaded data. Second, it tries to dispatch the paths to GPUs for parallel processing according to the topological order of the dependency graph of them. Many paths then converge along such an order after processing them for exactly once, getting lower reprocessing overhead. Third, a path scheduling strategy is further developed on each streaming multiprocessor to enable the privileged execution of the paths (e.g., the hot paths) with greater impacts on vertex state propagation for shorter convergence time according to vertex dependency. Experimental results show that our approach speeds up iterative directed graph processing by up to 3.54 times in comparison with the state-of-the-art systems.

CCS Concepts • **Computer systems organization → Parallel architectures**; • **Computing methodologies** → *Parallel computing methodologies*.

Keywords Iterative directed graph processing; GPU; convergence speed; data access cost; warp scheduling

ACM Reference Format:
Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An Efficient Path-based Iterative Directed Graph Processing System on Multiple GPUs. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304029

1 Introduction

Many iterative algorithms have been recently proposed to analyze directed graphs (e.g., protein interaction networks, social networks, and citation networks) for various applications [3, 12, 14, 22, 29, 31, 44, 56], because directed graphs are prevalent in the real world. It is usually time-consuming for them to handle the large-scale directed graph round by round until the states of the vertices are stable. Therefore, it is expected to reduce the execution time of these iterative directed graph algorithms to provide real-time results for the related applications.

* Yu Zhang, Xiaofei Liao (Corresponding author), Hai Jin, Haikun Liu, and Lin Gu are with National Engineering Research Center for Big Data Technology and System, Services 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.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6240-5/19/04...$15.00
https://doi.org/10.1145/3297858.3304029

With the recent advances in GPU accelerators, there is a growing interest to offload graph analytics to GPUs for real-time performance, due to its higher computing power and memory bandwidth than the CPU. Many systems, such as Medusa [54], CuSha [16], GraphReduce [36], Garaph [25], Gunrock [30, 46], and Groute [4], have been recently designed to handle large-scale graph by exploiting the powerful capacity of GPUs on a single PC for high cost-effectiveness. They support efficient iterative graph processing on GPUs through hiding communication cost, ensuring balanced load, improving data locality, and ensuring high GPU utilization ratio.

Despite of research efforts in the previous studies, there is a major challenge for iterative directed graph processing on the GPUs. Because existing GPU-based systems [4, 16, 25, 30, 36, 46, 54] take vertex/edge as the basic parallel processing unit over GPU cores, the vertices of each directed path may be concurrently handled by many GPU threads and update their states based on their precursors' stale states within each round of graph processing. As a result, the new state of each active vertex needs many rounds to work on its successors for the unawareness of update dependency[1] between vertices, and its stale state is used by many GPU threads to iteratively compute the states for other vertices, which are distributed over many or all graph partitions. Such a slow propagation of new vertex state finally induces frequent reprocessing of many partitions for existing GPU-based systems to repeatedly update their vertices' states, although few vertices may be active within the latter rounds. It not only wastes a long time to repeatedly handle the same vertices, but also incurs high cost to load them. It motivates us to design more efficient GPU-based system for faster convergence speed and lower data access cost of iterative directed graph processing.

We analyse the characteristics of vertex update dependency in iterative directed graph processing over GPUs, and observe that, a vertex can quickly propagate its recent state to the others along a directed path when the vertices of this path are sequentially and asynchronously handled along this path within each round. Based on this observation, we present DiGraph, a GPU-accelerated iterative directed graph processing system by using our path-based asynchronous execution model, which is fundamentally different from existing solutions. In particular, it represents a directed graph into a set of disjoint hot/cold paths, so as to provide an opportunity to efficient vertex state propagation and higher utilization ratio of the loaded data. The directed path is then taken as the basic parallel processing unit, where different paths are concurrently handled by GPU threads and the vertices of each path are sequentially handled by a single GPU thread along their order on this path within each round in an asynchronous way. In this way, vertex state propagation is able to be quickly done along the directed paths, getting faster convergence speed.

For lower reprocessing cost, a topological order is tried to be generated for the paths according to their dependencies and the paths are tried to be dispatched to GPUs for parallel processing according to such an order. Then, many paths never need to be reprocessed once they have been handled, due to no new vertex state from the other paths. On each *Streaming Multiprocessor* (SMX), an efficient path scheduling strategy is further designed to judiciously assign the processing order of its paths for less redundant work according to their importance on vertex state propagation. Compared with Gunrock [30, 46] and Groute [4], two cutting-edge GPU-based graph processing systems, experimental results show that DiGraph offers improvements of 2.25–7.39 and 1.59–3.54 times for iterative directed graph processing on four GPUs, respectively. Besides, when the number of GPUs increases from one to four, the graph processing time of DiGraph is reduced by 62.9%, more than 46.3% of Gunrock and 56.5% of Groute, indicating better scalability than them.

The paper has the following important contributions on GPU-based graph processing:

- It proposes a path-based asynchronous execution model to exploit our observed characteristics of update dependency to accelerate iterative directed graph processing on multiple GPUs.
- It also tackles several fundamental challenges for directed graph processing on multiple GPUs. Our techniques include: 1) a path-based directed graph partitioning method to provide an opportunity to faster state propagation and higher utilization ratio of GPUs (Section 3.2.1); 2) a storage scheme of directed graph for coalesced accesses and lower storage cost on GPUs (Section 3.2.1); 3) a dependency-aware path dispatching for multi-GPUs for less redundant work and higher utilization ratio of G-PUs (Section 3.2.2); 4) a path processing method for faster convergence speed, higher parallelism of GPU threads, and lower communication cost (Section 3.2.2); and 5) a path scheduling scheme on SMX for faster state propagation (Section 3.2.3).
- It gives an evaluation to demonstrate the efficiency of DiGraph on a platform with multiple GPUs.

The remainder is organized as follows: Section 2 describes the challenges of existing GPU-based solutions and our motivations. Section 3 presents our approach and the details of DiGraph, followed by an evaluation in Section 4. Section 5 gives a survey of related work. Finally, we conclude this paper in Section 6.

[1] Each directed edge $<v_i,\ v_j>$ induces an update dependency between v_i and v_j, indicating that v_j updates its own state based on the state of v_i.

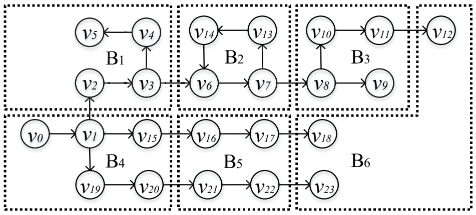


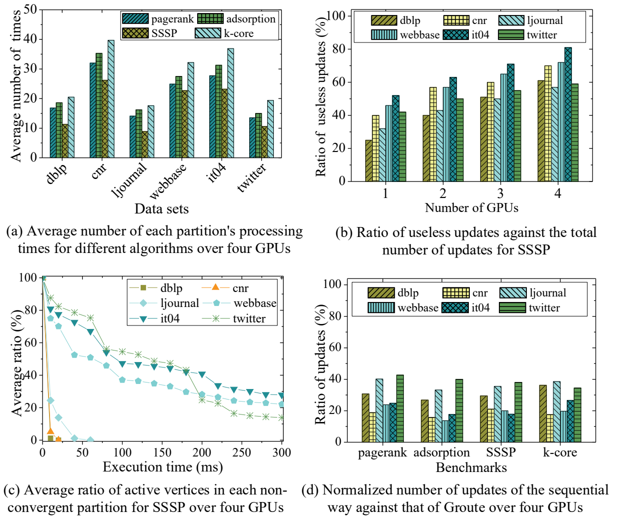
Figure 1. An example to illustrate the inefficiency of existing GPU-based execution models for iterative directed graph processing, where the directed graph is divided into six partitions, i.e., B_1, B_2, B_3, B_4, B_5, and B_6.

2 Challenges and Motivation

Recently, a single PC is usually equipped with multiple GPUs (each contains multiple SMXs) and these GPUs are connected through PCI-Express or NVLink. A GPU program has host code and device code (GPU kernel). Each GPU kernel is *Single Instruction Multiple Threads* (SIMT) program. The data is dispatched from the host to the global memory of GPU for its threads to handle. The GPU threads are grouped into warps. A SMX holds a certain number of warps and its warp scheduler assigns their processing order in a round-robin order by default. The threads on each SMX communicate with each other through its on-chip shared memory.

Inefficiency of Existing Solutions. For iterative directed graph algorithms, each vertex needs to repeatedly update its state according to its precursors' states until convergence. However, when it is executed over GPUs, most vertices and their precursors are concurrently handled by GPU threads and may update their states according to their precursors' stale states in each round of graph processing. As a result, the new states of active vertices are slowly propagated to others along the directed paths, and many vertices have iteratively calculated their states based on stale states of these vertices and need reprocessing, although using existing synchronous/asynchronous GPU-based solutions [4, 30, 46]. Much time is wasted to repeatedly handle the same vertices and high cost is also generated to load them.

Take Figure 1 as an example and assume that one active vertex, e.g., v_2, exists in the current iteration. With existing synchronous GPU-based methods [30, 46], after one round of graph processing, the new state of v_2 can only be propagated to its direct neighbors, because each new vertex state cannot be used by other vertices in the same round. In order to propagate the new state of v_2 to v_5, three rounds of graph processing are needed. With existing asynchronous GPU-based solutions [4], although the latest state of each vertex is allowed to be immediately used by its neighbors, the vertices of a directed path may be concurrently handled by different GPU threads within each round. Because the states of already-processed vertices can only be updated in the next round, it also suffers from slow state propagation.



(a) Average number of each partition's processing times for different algorithms over four GPUs

(b) Ratio of useless updates against the total number of updates for SSSP

(c) Average ratio of active vertices in each non-convergent partition for SSSP over four GPUs

(d) Normalized number of updates of the sequential way against that of Groute over four GPUs

Figure 2. Evaluation of Groute over GPUs

For example, when asynchronously executing SSSP (v_1 is the source vertex) over GPUs, the vertices in B_1 of Figure 1 may be concurrently processed by different GPU threads of a warp. Then, v_5, v_4, and v_3 have been handled before receiving the new state (the shortest distance from v_1 to v_2) of v_2 within the current round. v_5, v_4, and v_3 have to be reprocessed in the next several rounds to update their states according to the new state of v_2. Because there are three hops for the state propagation from v_2 to v_5, the partition B_1 needs three rounds of processing for convergence. Furthermore, after the processing of B_1, the partitions B_2, B_3, and B_6 need to be reprocessed to update their vertex states according to the received new vertex states from B_1, although they have been handled. It exacerbates the above challenge.

To demonstrate it, Groute [4] is evaluated over a platform with four GPUs. The details of the hardware platform and the benchmarks are the same as those described in Section 4. From Figure 2(a) and Figure 2(b), we observe that many partitions of Groute need frequent reprocessing, because each vertex usually reads stale states of its neighbors for useless update before updating these stale states in the same round. It is more serious on the platform with more GPUs. Besides, as depicted in Figure 2(c), only a few vertices of non-convergent partitions are active, indicating low GPU utilization ratio as well, where all vertices are initially set active.

Our Observations. Figure 2(d) experimentally shows the number of vertex updates required by the sequential execution of iterative directed graph algorithm, where all vertices are tried to be sequentially and asynchronously handled by a thread according to the topological order of the directed graph. In the experiments, we have the following two observations in iterative directed graph processing. They motivate our design to fully exploit update dependency between vertices for faster convergence and much lower reprocessing overhead on GPUs.

Observation 1: A vertex's new state is able to work on the other ones within a single round if they are handled sequentially along their order on a directed path in an asynchronous way within the round. For example, let the vertices of B_1 (see Figure 1) be sequentially and asynchronously handled in the order of v_2, v_3, v_4, and v_5 for one round. Then, the new state of v_2 can be propagated to v_3. In addition, it also can be aggregated by v_3 and works on v_3, when v_3 is handled within the same round. Similarly, the newly generated state of v_3, calculated based on the new state of v_2, is also able to sequentially work on v_4 and v_5, when v_4 and v_5 are processed in the same round. That is to say, the new state from v_2 can reach v_3, v_4, and v_5 using a single round instead of three rounds, and B_1 needs much fewer vertex state updates to converge.

Observation 2: Lots of directed edges do not constitute a cycle[2]*, which allow to further reduce redundant processing.* For example, v_8 should be handled after the convergence of v_7. By such means, v_8 will never receive new vertex state from the other vertices, because the directed edge from v_7 to v_8 is not in any cycle. It also indicates that the partition B_3 never needs to be reprocessed again, if it is handled after the convergence of the partition B_2. Thus, in Figure 2(d), for different iterative graph algorithms, 18.1%-23.3%, 31.8%-37.4%, 9.5%-13.9%, 27.2%-36.5%, 17.6%-22.8%, and 8.2%-13.3% of all vertices only need one update to converge over dblp, cnr, ljournal, webbase, it04, and twitter, respectively, because the directed edges on the paths between two vertices of different *Strongly Connected Components* (SCCs) do not constitute cycle, although the vertices of the giant SCC occupy 69.4%, 34.4%, 78.0%, 45.6%, 72.3%, and 80.3% of all vertices for the graphs, respectively. Note that all vertices of directed acyclic graph only need one update when they are handled along their topological order, because all edges are not in any cycle.

3 Overview of DiGraph

Based on the observations, we propose an efficient GPU-based iterative directed graph processing system, called DiGraph. It represents a directed graph into a set of disjoint directed paths and takes the path as the basic parallel processing unit. This is fundamentally different from existing solutions, which take vertex/edge as the basic parallel processing unit. In DiGraph, different paths are concurrently handled by GPU threads. The vertices of each path are sequentially handled by a single GPU thread along their order on this path within each round and each vertex's new state is asynchronously sent to its successors for the updating of their states within the same round, getting faster vertex state propagation. There, the high-degree vertices, i.e., *hot-vertices*, are tried to be put together into hot paths, aiming to enable efficient vertex state propagation and higher utilization ratio of the loaded data. To reduce the overhead of repeated processing of the paths, it constructs a *Directed Acyclic Graph* (DAG) for the paths based on their dependencies, and tries to assign them to GPUs for parallel processing according to the topological order of the DAG, where each node of the DAG sketch represents a subset of paths. A path scheduling strategy is further used on each SMX to arrange the processing order of its paths based on their importance on state propagation, enabling the privileged execution of the most important paths (e.g., hot paths) for less redundant work.

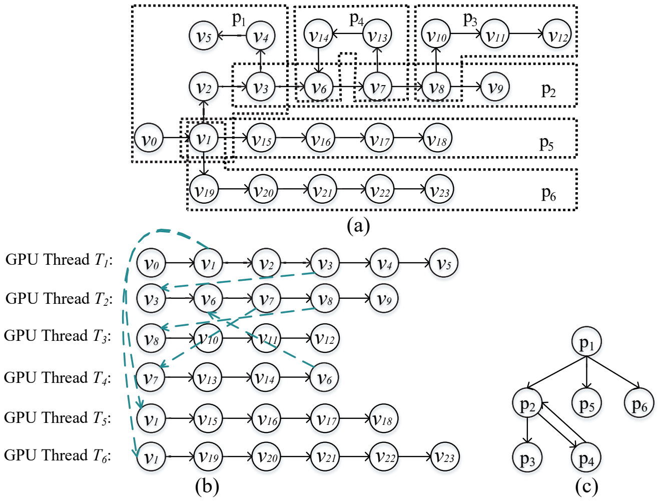


Figure 3. An example to show how to divide the directed graph into directed paths and get the dependency graph for them: (a) the directed graph is divided into six paths; (b) the update dependencies between the paths which are concurrently processed by different GPU threads; (c) the generated dependency graph of the paths, which indicates that the processing order of the paths is expected to be p_1, p_2, p_4, p_5, p_6, and p_3.

3.1 Path-based Asynchronous Execution

Let G=(V, E) denote a directed graph, where V=$\{v_l | 0 \leq l < n\}$ is a set of vertices, E=$\{<v_l, v_m> | v_l \in V \wedge v_m \in V\}$ is a set of directed edges, and n is the number of vertices. According to our observation, the vertices of each directed path can quickly propagate their new states to others only when they are asynchronously handled along their order on this path within each round in a sequential way, due to the update dependencies of them. In order to exploit such a characteristic of update dependency for faster convergence, in our model, the graph G is represented by a set of disjoint directed paths (no intersected edge), i.e., G=P=$\cup_{(x \in V \wedge y \in V)} Path(x, y)$, where $Path(x, y)$ is an ordered sequence of connected

[2] A cycle, e.g., $v_6 - v_7 - v_{13} - v_{14} - v_6$ of Figure 1, is an ordered sequence of connected directed edges such that any vertex on it is able to reach itself.

directed edges from the vertex x to y. There, it is expected that $\forall$ $Path(x,y) \in P \wedge \forall Path(x',y') \in P$ s.t. $Path(x,y) \cap Path(x',y') \subseteq \{x,y,x',y'\}$, making sure that more paths are not dependent on each other for less reprocessing cost. An example is given in Figure 3(a). Otherwise, for any two paths with intersected vertices, they are dependent on each other and need reprocessing.

After that, it takes the directed path as the basic parallel processing unit and the GPU threads concurrently deal with different paths. The vertices of each directed path are sequentially handled by a single GPU thread along their order on this path (e.g., v_0, v_1, $\cdots$, v_5 for the path p_1 which is handled by the GPU thread T_1 of Figure 3(b)) within each round in an asynchronous way, where the vertices are also stored along such an order for coalesced accesses. The new state of each vertex (e.g., v_1) is immediately used to update the state of its direct successor (e.g., v_2) on the same path in the same round. By such means, the states of the vertices on each directed path can reach the others along this path within one round, no matter how long the path is. The maximum number of graph processing rounds for one vertex state propagation to another equals to the number of paths traversed by this propagation, which is usually much fewer than that of existing solutions, i.e, the diameter of the graph. For example, as described in Figure 3(b), our approach uses three rounds to propagate the new state of v_0 to v_{12} along three paths, i.e., p_1, p_2, and p_3, indicating faster convergence speed than existing solutions. Note that the state propagation between paths is done through state synchronization of vertex replicas (e.g., three replicas of the vertex v_1), where it takes partition as the synchronization unit and is conducted in batches for low overhead (Section 3.2.2).

However, some directed paths are dependent on others and may face reprocessing when they are concurrently handled by different GPU threads (Figure 3(b)), because the states of their vertices may be updated based on stale vertex states of the other paths. To avoid the repeated processing of some paths, it generates an effective dependency graph G' for the paths, i.e., G'=(V', E'), where V'=$\{p_i|\ p_i \in P\}$ and E'=$\{<p_i,\ p_j>|\ \exists\ p_i \in P \wedge \exists\ p_j \in P$ s.t. $v_l \in (p_i \cap p_j) \wedge <v_n,\ v_l> \in p_i \wedge <v_l,\ v_m> \in p_j\}$. Figure 3(c) illustrates how to get the dependency graph for the paths of a directed graph. After that, it can get a DAG sketch G'' of G' by contracting SCCs of this dependency graph into nodes, i.e., G''=(V'', E''), where V''=$\{SCC_m|\ SCC_m \subseteq G' \wedge 0 \leq m < |S|\}$, $|S|$ is the number of SCCs in the graph G', E''=$\{<SCC_x,\ SCC_y>\ |\ \exists\ p_i \in SCC_x \wedge \exists\ p_j \in SCC_y$ s.t. $p_i \rightarrow p_j\}$, and $p_i \rightarrow p_j$ denotes that p_i is able to reach p_j through a directed path of G'.

Each vertex, i.e., SCC_m, or called *SCC-vertex*, of the DAG represents a set of paths and each edge in the DAG describes the update dependency between two SCC-vertices, which allows the paths to be handled according to the topological order of this DAG sketch. The set of paths represented by each SCC-vertex are tried to be asynchronously dispatched to GPUs for parallel processing, when the precursors of this SCC-vertex are inactive. Then, due to no new vertex state from the other ones, many paths never need to be handled again, sparing redundant reprocessing. Note that, if there are idle SMXs (e.g., the number of schedulable paths is less than that of the idle GPU cores), the successive paths represented by the active SCC-vertex with the least number of active precursors are also allowed to be handled in advance to get vertex states nearer to the final ones by fully exploiting high parallelism of GPUs.

3.2 Parallel Processing over Multiple GPUs

We firstly show the details of parallel preprocessing of directed graph on the CPU, then show how to concurrently handle it on multiple GPUs. Note that we also use the NCCL library[3] to create a ring topology over the bus for efficient multi-GPU collective communication as Groute [4], and the communication between multiple GPUs is done through the host memory. Graph algorithms on DiGraph are implemented by the APIs of the popular Gather-Apply-Scatter programming model [8].

3.2.1 Graph Representation and Storage

We firstly describe how to efficiently represent and store the directed graph.

Path-based Graph Partitioning. A few high-degree vertices, i.e., hot-vertices, need more rounds of processing to converge, because most vertices' states pass through them to reach others for power-law property [8] of real-world graph. Thus, when dividing a graph into paths, hot-vertices are expected to be put together into several hot paths (e.g., p_2 in Figure 3(a)), while the others are put into cold paths. It brings two advantages. First, it enables the privileged execution of hot path, which serves as fast track for most state propagations. Second, fewer easily convergent vertices (e.g., v_9) are loaded along with the frequently processed hot-vertices (e.g., v_6) as the vertices on the same path, sparing the consumption of memory bandwidth and shared memory of SMX.

We employ a parallel method to decompose the directed graph. In detail, it firstly divides the graph into several subgraphs and evenly assigns them to CPU threads to divide them into paths. As shown in Algorithm 1, each CPU thread repeatedly takes the vertex (i.e., v_{root}) with unvisited local edges as the root, and travels the edges of its subgraph in a depth-first order to divide this subgraph into paths, until all edges are visited, where d=0 for each

[3] http://www.github.com/NVIDIA/nccl/.

Algorithm 1 Path-based Graph Partitioning on CPU

```
1: procedure GRAPHP($v_{root}$, $p$, $d$)
2:     Set the vertex $v_{root}$ as visited
3:     if It has unvisited local edges $\wedge$ $d<D_{MAX}$ then
4:         /*Get $v_{root}$'s successors in local subgraph*/
           $S_{suc}$ ← GetLocalSuccessors($v_{root}$)
5:         Sort($S_{suc}$) /*Sort them based on degrees*/
6:         for each $v \in S_{suc}$ do
7:             if <$v_{root}$, $v$> is unvisited then
8:                 Set the edge <$v_{root}$, $v$> as visited
9:                 /*Insert into the queue pointed by $p$*/
                   Insert($p$, <$v_{root}$, $v$>)
10:                if $v$ is unvisited then
11:                    GRAPHP($v$, $p$, $d$+1)
12:                else
13:                    Set the vertex $v$ as an inner one
14:                    NewPath($p$)
15:                end if
16:            end if
17:        end for
18:    else
19:        NewPath($p$)
20:    end if
21: end procedure
```

traversal. Note that the depth, i.e., d, (its maximum value is D_{MAX}=16 by default) of traversal is bounded (see line 3) to make the lengths of generated paths not too skewed. When there is a set of neighbors to be visited, the one with the highest degree is first chosen aiming to divide the edges between high-degree vertices together into a hot path (see lines 5 to 17). The recursively visited edges of each traversal by a CPU thread are successively added into the same edge queue (pointed by a pointer p) to constitute paths (see lines 9 and 11). Note that the edge queue is shared by its each traversal and also has an auxiliary array to indicate the range of its each path by recording the offset of the first vertex of its each path in the queue. When a new path will be generated (see lines 14 and 19), it only needs to record the edge queue's offset to store this path's first vertex. In this way, the edges are divided into a set of disjoint hot/cold paths. The vertices are also stored following their original order on the path, ensuring efficient state propagation along the path and high locality of vertex processing. Note that each thread only divides its local subgraph (see line 4), incurring no communication cost. The generated paths are ensured to satisfy the constraint defined in Section 3.1 for less reprocessing cost (see line 14).

During the above stage, some short paths may be generated by different CPU threads. However, the average length of the paths is obviously expected to be maximized for faster convergence speed. Thus, after that, the paths are rechecked to merge the short ones in a head-to-tail way for larger average length. To still maintain the constraint for them, for two paths, if both in-degree and out-degree of their intersected vertex are more than one, they are merged only when their intersected vertex is not an *inner vertex* (which is not the head/tail vertex of a path and is identified as line 13) of the other paths.

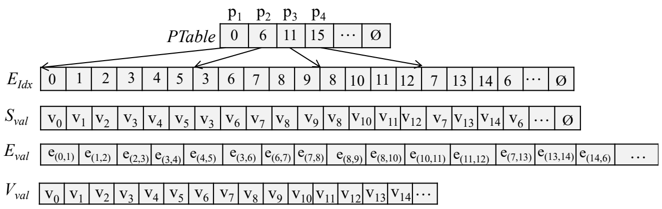

p1 p2 p3 p4
PTable 0 6 11 15 ··· Ø
E_{Idx} 0 1 2 3 4 5 3 6 7 8 9 8 10 11 12 7 13 14 6 ··· Ø
S_{val} v_0 v_1 v_2 v_3 v_4 v_5 v_3 v_6 v_7 v_8 v_9 v_8 v_{10} v_{11} v_{12} v_7 v_{13} v_{14} v_6 ··· Ø
E_{val} $e_{(0,1)}$ $e_{(1,2)}$ $e_{(2,3)}$ $e_{(3,4)}$ $e_{(4,5)}$ $e_{(3,6)}$ $e_{(6,7)}$ $e_{(7,8)}$ $e_{(8,9)}$ $e_{(8,10)}$ $e_{(10,11)}$ $e_{(11,12)}$ $e_{(7,13)}$ $e_{(13,14)}$ $e_{(14,6)}$ ···
V_{val} v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} ···

Figure 4. Representation of directed graph on GPUs

For less reprocessing cost of paths, a dependency graph is also created for them when dividing the graph at the above stage. A DAG sketch is then generated for these paths via parallelly contracting each SCC of this dependency graph to SCC-vertex. In detail, the dependency graph is divided into subgraphs and each CPU thread uses tarjan algorithm [40] to find local SCCs of each subgraph to generate local DAG via contracting its local SCCs to SCC-vertices. This step only needs to travel the dependency graph of paths for exactly once [40]. Then, tarjan algorithm is used again to further contract the graph consisting of local DAG sketches into the global DAG sketch, where each local SCC is taken as a vertex.

The paths are then selectively assigned to partitions. The highly-connected paths (especially the paths belonging to the same SCC-vertex or highly-connected SCC-vertices which can be concurrently handled) are tried to be assigned to the same partition for higher utilization ratio of the loaded data and fewer idle GPU threads, according to the dependency graph of them. Note that the short paths with the replicas of the same high in-degree vertex are also tried to be put together to be executed over the same GPU thread or fewer ones, so as to reduce the impacts of write contention among GPU threads. In addition, the hot paths are also tried to be put together for higher utilization ratio of GPU resource, while the remaining cold paths are inserted into the other partitions. Otherwise, more paths of a partition may be inactive when this partition is handled on a SMX, because the cold paths may need fewer rounds to converge than the hot ones.

Storage of Directed Paths. To efficiently store directed graph over GPUs, four arrays are employed. E_{Idx} is used to sequentially store the indexes of source vertices for the directed edges of each directed path along their order on this path, where each directed edge can be represented by two successive items of E_{Idx}. It also means that less space is required by our approach to store the directed edges than shard-based approach [16, 25], which is demonstrated to be the best graph representation method. The values of the source vertices and the edges are stored in S_{val} and E_{val}, respectively. Besides, we also maintain a separate array named V_{val} for quick access to the most recent state value of each vertex. *PTable* is also established to describe the information of paths and contains a field to store the index of each path's first vertex. Note that the paths of a partition are stored in successive items of *PTable* and E_{Idx}. Two successive items of *PTable* indicate the range of a path. Figure 4 illustrates how to store the directed graph described in Figure 3(a). In this way, the threads of a warp (assigned with a partition) can read consecutive data residing in the global memory, ensuring coalesced accesses to them.

3.2.2 Dependency-aware Path Processing

For some directed graphs, the parallel execution of the paths may suffer from poor performance on GPUs. First, some SCC-vertices (or called *hub SCC-vertices*) have much more out-edges than the others. The delayed processing of their paths may induce under-utilization of many GPU cores, because many paths are dependent on them and need to access their results. In addition, the hub SCC-vertex may be a giant one, which occupies a major proportion of all graph paths. It needs much longer time than the other SCC-vertices to repeatedly process many highly-connected paths for much more rounds in an iterative way, exacerbating the above problem. Second, when processing the paths, the GPU threads may face low parallelism and high communication cost, due to the skewed lengths of the generated paths and high synchronization cost (e.g., heavy write contention and interleaved communication). In the following part, we show how to tackle these challenges.

Dependency-aware Path Dispatching for Multi-GPUs. For efficient parallel processing, it firstly divides the DAG sketch into layers and assigns each SCC-vertex with a layer number, where the SCC-vertices at each layer only depend on the SCC-vertices at the lower layer. After that, the paths are tried to be asynchronously dispatched to GPUs for parallel processing layer by layer. In detail, when some SMXs become idle, the paths represented by the active SCC-vertex with the smallest layer number are first dispatched from the host to the idle SMXs for parallel processing. The paths represented by some SCC-vertices (e.g., the giant SCC-vertex) may be concurrently executed over SMXs of multiple GPUs, while the paths represented by several small SCC-vertices may be executed over a SMX. For higher parallelism, the processing order of the SCC-vertices at the same layer is arranged in descending order according to the total number of paths in their successive active SCC-vertices. Then, more successive paths are able to be handled when some SMXs become idle.

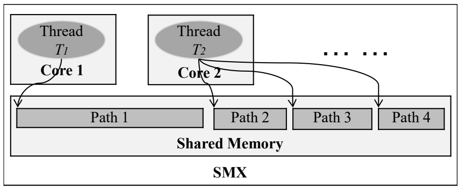


Figure 5. Concurrent path processing over each SMX

The above path dispatching from the host memory to the GPU memory is done in batches for low traffic. To overlap memory copy and kernel execution, multiple streams are created for the transfer of paths using Hyper-Q of GPU. The number of streams is expected to be $N_m = \frac{M_G}{S_b}$, where M_G is the global memory size of a GPU and S_b is the size of each batch. Besides, for the running paths, their successive paths are also transferred to the related GPU in advance. By such means, GPU kernels can be immediately launched to handle these successors on SMXs without the delay to fetch them from the host memory, when the running paths are inactive.

For lower communication cost, the results of each path for its successors are tried to be buffered in the global memory of the related GPU. The paths represented by each active SCC-vertex are always tried to be dispatched to the GPU with the most number of its direct precursors. By such means, the processing of its paths needs less communication cost to access the results of those precursors. When the global memory of a GPU is not enough to buffer the results, some need to be transferred to the other GPUs or the host memory. In order to maximize the performance, the buffered results of the paths represented by a SCC-vertex are swapped out of a GPU when this SCC-vertex has the least number of active direct successors on this GPU. Note that the results of the paths represented by a SCC-vertex may never need to be accessed and can be written back to the host memory, when all of its direct successors are inactive.

Work Stealing for Balanced Load between S-MXs. Some SMXs may be released by early convergent paths at runtime. Thus, the paths of the overloaded SMXs should be dynamically stolen to the free SMXs so as to ensure load balancing between SMXs. In detail, the set of suspended paths (e.g., the ones represented by the giant SCC-vertex) are allowed to be evenly divided into several subsets. After that, these subsets are asynchronously stolen to the free SMXs to assist their processing, where the paths are always tried to be first stolen to the free SMX on the local GPU. Then, the suspended paths can quickly converge and their successive paths can be handled as early as possible, due to more parallelism and higher locality.

Asynchronous Path Processing on SMX. After that, the GPU threads on each SMX concurrently handle its paths. However, the lengths of the generated paths may be skewed. Because the GPU threads of a warp execute the same instruction in lock-step fashion, when different-sized paths are assigned to be handled by GPU threads on the same SMX, it eventually leads to underutilization of this SMX for the imbalanced load among these GPU threads, although it can get balanced load between different SMXs of different GPUs as the above discussed. Thus, the paths on each SMX are tried to be evenly assigned to its GPU threads to ensure that the number of the edges handled by its each GPU thread is almost equal. It also means that some GPU threads will be assigned with several short paths when a thread of the same SMX is assigned with a long path for processing. Figure 5 gives an example to illustrate it. Note that the thread assigned with the hot path does not have a heavier burden than the other ones, although more propagations are conducted on hot paths, because vertex states are aggregated on hot-vertices before their propagations. The number of edges determines each path's load.

The vertices of each directed path are sequentially handled by a GPU thread along their order on this path in each round. The newly calculated state of each vertex is allowed to be immediately used to update the states of its successors in the same round. Note that a vertex may have multiple replicas, which are distributed over paths. One of the replicas is nominated as the *master* (see V_{val} of Figure 4), while the others are regarded as the *mirrors* (see S_{val} of Figure 4). Each mirror always sends its new state to its master for state synchronization. Other mirrors are able to pull the most recent state from the master for the state updates of their local successors.

However, heavy write contention may still exist between GPU threads due to many atomic updates of the state of the same master. Thus, it creates a *proxy vertex* in the shared memory of each related SMX for each vertex with high in-degree, to accumulate the pushed new states of its local mirrors on the same SMX. The accumulated results on the proxy vertex are allowed to be used by these local mirrors to update their successors in the same round, and are sent to the master when the partition assigned to this SMX has been handled. When synchronizing states between vertex replicas, some partitions need to be repeatedly accessed and high communication cost is generated, because the messages (containing the pushed new vertex states) to the vertex replicas on the same partition may be interleaved with the messages to the ones on the other partitions. For lower overhead, after the processing of each partition, the messages generated on the related SMX are arranged according to the IDs of the destination partitions before pushing them, and the ones to the same partition are propagated together in batches. Then, fewer partitions are loaded and less communication cost is required for the state updates of the replicas, because many updates become successive accesses to the same partition.

3.2.3 Path Scheduling over SMX

On each SMX, the paths are assigned to be handled by the warp scheduler in a round-robin order by default. However, different paths have various impacts on state propagation and incur different volume of useless work, because of the skewed power-law degree distributions of the real-world directed graphs. First, more vertex state propagations are done through the path (e.g. the hot path) with higher average vertex degree. Second, the number of active vertices are also skewed on different paths. Thus, inefficient processing order of paths on a SMX leads to more redundant work and the vertices are activated more times to update themselves based on their neighbors' stale states, due to slow propagations of more active vertices' recent states. Efficient soft path scheduling strategy is needed for SMX to assign the processing order of its paths based on their importance on state propagation, so as to reduce redundant processing.

It gives each path p a soft priority $Pri(p)=\alpha \cdot \overline{D}(p) \cdot N(p)-L(p)$, where $\overline{D}(p)$, $N(p)$, and $L(p)$ are the average vertex degree, the number of active vertices, and the layer number of p, respectively. $\alpha = \frac{1}{\overline{D}_{max} \cdot N_{max}}$ is the scaling factor set by the runtime system at graph preprocessing time to ensure that the path with the smallest value of $L(p)$ is given the highest priority, where $\overline{D}_{max}$ and N_{max} are the maximum average vertex degree and the maximum number of vertices of any path, respectively. The values of $L(p)$, $\overline{D}(p)$, and the initial value of $N(p)$ are gotten at preprocessing time, while $N(p)$ is incrementally updated at the execution time. When a SMX becomes idle, the set of paths with the largest values of $Pri(p)$ are first processed on it and the low-priority paths are deferred to a later stage, so as to shorten the total execution time by reducing the amount of redundant work. Otherwise, the processing of low-priority paths iteratively activates more useless updates, which will contend with the high-priority paths for GPU resource.

4 Experimental Evaluation

The hardware platform used in our experiments is a system with four GPUs. Each of them is NVIDIA TESLA K80, which totally has 26 SMXs (4992 cores) and 24 GB on-board memory. On the host side, there are 64 GB main memory and four 8-core 2.60 GHz Intel Xeon E5-2670 CPUs, where each CPU has two 8 GT/s QPI link and PCI Express 3.0 lanes operates at 16x speed. The program is compiled by GCC 4.9 and CUDA 8.0 with the -O3 flag. In our experiments, four iterative directed graph algorithms are employed as benchmarks: (1) pagerank [29]; (2) adsorption [3]; (3) SSSP [28]; (4) k-core [14]. All graph algorithms execute until convergence. Six real-world directed graphs [1], i.e., dblp-2010, cnr-2000, ljournal-2008, webbase-2001, it-2004, and twitter-2010 are used. Table 1 summarizes their characteristics.

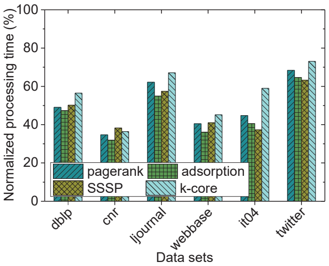


Figure 6. Graph processing time of DiGraph against DiGraph-t

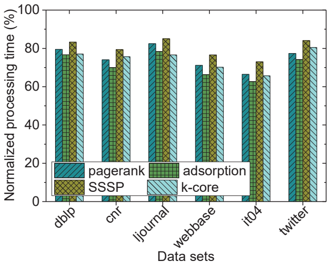


Figure 7. Graph processing time of DiGraph against DiGraph-w

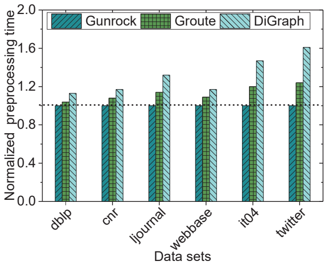


Figure 8. Normalized graph preprocessing time of different systems

Table 1. Data Sets Properties

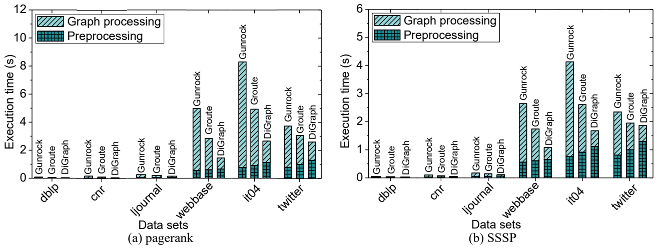
(A_{Deg} and A_{Dis} denote the average degree of all vertices and the average distance between any two vertices, respectively.)

Data sets	$\#Vertices$	$\#Edges$	A_{Deg}	A_{Dis}
dblp	326,186	1,615,400	4.952	7.35
cnr	325,557	3,216,152	9.879	17.45
ljournal	5,363,260	79,023,142	14.734	5.99
webbase	118,142,155	1,019,903,190	8.633	17.19
it04	41,291,594	1,150,725,436	27.868	15.04
twitter	41,652,230	1,468,365,182	35.253	4.46

To understand the advantages of DiGraph, two other versions of DiGraph are also implemented and evaluated. *DiGraph-t* employs the traditional asynchronous execution model [4] instead of our path-based asynchronous execution model. *DiGraph-w* uses our asynchronous execution model yet without using our path scheduling strategy. Besides, DiGraph is further compared with Gunrock (version 0.4) [30, 46] and Groute (version 1.0) [4], which are the cutting-edge graph processing systems on multiple GPUs using synchronous and asynchronous model, respectively. Note that, for fairness, the performance of both Gunrock and Groute is tuned to be the best. All experiments (except Figure 8, Figure 16, and Figure 17) are evaluated over four GPUs.

4.1 Performance of DiGraph

We firstly show the superiority of our path-based asynchronous graph processing model. Figure 6 gives the normalized graph processing time of DiGraph compared with DiGraph-t. We can find that DiGraph always gets a better performance than DiGraph-t. Take pagerank as an example. DiGraph reduces graph processing time of DiGraph-t by about 65.3% over cnr. We find that the higher convergence speed of DiGraph mainly comes from much fewer vertex state updates. It is because that, in the traditional asynchronous model, the stale vertex states are usually used by multiple GPUs to compute new states for many vertices. Exceptionally high redundant work is generated to recalculate the new states for all these vertices when the stale states are overwritten. In our path-based asynchronous model, vertex state can be more quickly propagated along directed paths and each vertex is also tried to be handled after the convergence of all its precursors. Consequently, it is able to spare many redundant vertex updates.



(a) pagerank

(b) SSSP

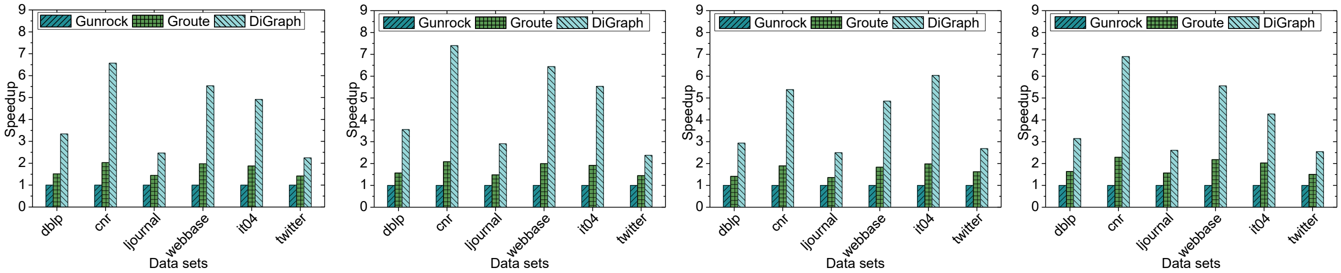
Figure 9. Execution time breakdown of different systems

Figure 7 depicts the normalized graph processing time of DiGraph against DiGraph-w so as to evaluate the performance of our path scheduling strategy. As seen in the figure, our scheduling scheme is able to effectively reduce the graph processing time. For example, in comparison with DiGraph-w, DiGraph reduces graph processing time of pagerank by 33.5% over it04. The main reason is that our path scheduling strategy enables much fewer redundant vertex updates by first handling the paths which are important to vertex state propagations.

4.2 Comparison on Multiple GPUs

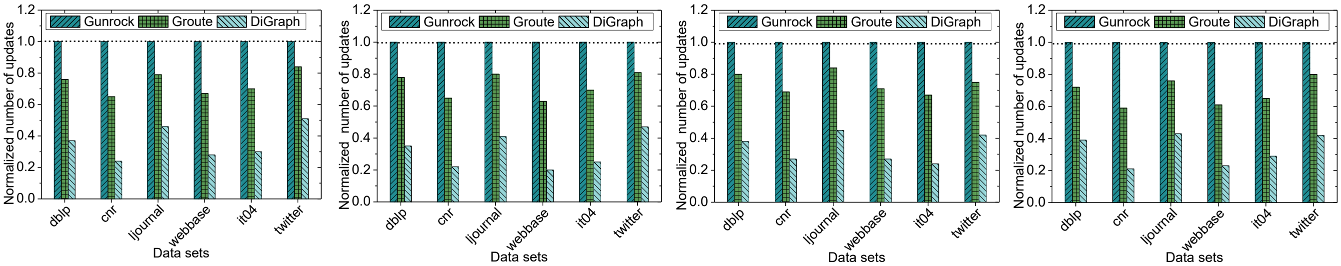
Existing systems need much cost to preprocess graph on CPU for less total execution time by ensuring balanced load and lower communication cost between multiple GPUs. We firstly evaluate the preprocessing time of Gunrock, Groute, and DiGraph against that of Gunrock. From Figure 8, we observe that DiGraph needs slightly more preprocessing time than the other solutions, where the preprocessing time of Gunrock is 0.015, 0.032, 0.049, 0.561, 0.762, and 0.809 seconds for dblp, cnr, ljournal, webbase, it04, and twitter, respectively. It is because additional overhead is required by DiGraph to concurrently divide subgraphs into directed paths by traversing the original graph for exactly once and to construct the DAG sketch for these paths by traversing the dependency graph once, where the number of vertices of the dependency graph is about 3.4%-9.1% of the original graph. For example, for dblp, the overhead of DiGraph is about 13% and 8.6% more than Gunrock and Groute, respectively. Note that DiGraph needs more preprocessing time for it04 and twitter because the ratio of the dependency graph against the original graph is higher.

However, as shown in Figure 9, for DiGraph, it brings significant benefits from two aspects as we will see in later experiments. First, it spares much more rounds



(a) pagerank (b) adsorption (c) SSSP (d) k-core

Figure 10. Speedup of different systems against Gunrock over four GPUs



(a) pagerank (b) adsorption (c) SSSP (d) k-core

Figure 11. Normalized number of updates of different systems against Gunrock over four GPUs

(i.e., much more vertex updates) for iterative directed graph algorithms, which usually traverse and handle the original graph for hundreds of rounds for convergence. Second, it ensures lower communication cost due to fewer updates and higher utilization ratio of the loaded data.

After that, the graph processing time of Gunrock, Groute, and DiGraph is evaluated. Figure 10 describes their speedups against that of Gunrock. As shown in the figure, Groute gets a better performance than Gunrock for much lower synchronization cost. For example, for pagerank, Groute achieves a performance improvement of 2.03 times in comparison with Gunrock over cnr. In addition, we also observe that Groute performs better over cnr than twitter. It indicates that the asynchronous execution model is much more suitable to the graph with longer diameter for no barrier between iterations. However, as observed in the experiments, Groute suffers from serious overhead to repeatedly handle many partitions and high communication cost to transfer much unnecessary data between GPUs, due to no consideration of vertex update dependency. For example, the graph processing time of adsorption on Groute is about 3.54 times that of DiGraph over cnr. DiGraph is able to get performance improvements of 2.25–7.39 and 1.59–3.54 times in comparison with Gunrock and Groute, respectively.

To demonstrate the above discussions, the number of updates for different graph algorithms to converge is firstly evaluated over different systems. Figure 11 shows the normalized number of vertex state updates of them in comparison with that of Gunrock. In this figure, we make two observations.

First, Groute needs to handle more vertex state updates than DiGraph, although it needs fewer updates than Gunrock to converge. Taking adsorption over it04 as an example, the number of updates of DiGraph is only 35.7% of that of Groute. Such a few number of updates of DiGraph mainly comes from faster vertex state propagation along the directed paths and an efficient processing order of paths by exploiting the update dependencies of them. Note that most graphs can get great benefits from the DAG-based optimization, because the processing of each small SCC may repeatedly incur the useless processing of the other SCCs (including the giant SCC) in existing solutions, although the giant SCC (occupies 3.5%-89.0% of all paths for different graphs) in the dependency graph may be large.

Second, from Figure 11, we also observe that DiGraph gets much better performance on the directed graph with longer average distance. For example, for PageRank, the number of updates needed by DiGraph is only 36.9% of Groute over cnr, while the ratio is 60.7% over twitter. It is because that both Gunrock and Groute need much more rounds to propagate vertex states for convergence when the average distance between any two vertices is longer. However, the number of rounds required by DiGraph is determined by the average number of paths traversed by its vertex state propagations. Its value is usually much fewer than the number of rounds needed by both Gunrock and Groute when the average distance is longer. Assume two opposite extreme scenarios. If the diameter of the graph is one, the performance of DiGraph degenerates to that of DiGraph-t, i.e., its efficiency is the same as the traditional one. If the graph only contains one path, DiGraph only needs to handle the graph once, while the number of rounds of both Gunrock and Groute may be equal to the length of the path. Note that the

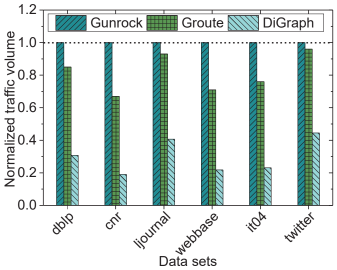


Figure 12. Normalized traffic volume of pagerank on four GPUs

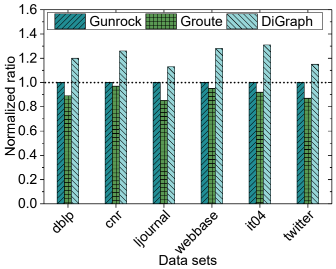


Figure 13. Normalized utilization ratio of loaded data against Gunrock

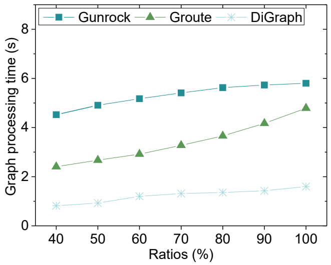


Figure 14. Impacts of the ratio of bi-directional edges

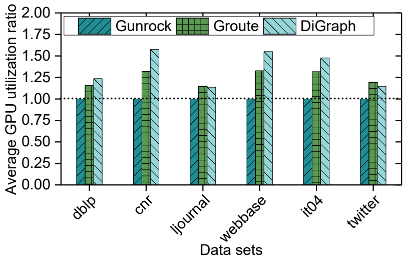


Figure 15. GPU utilization ratio of pagerank

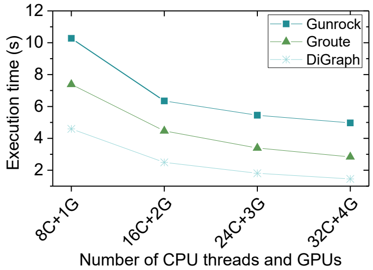
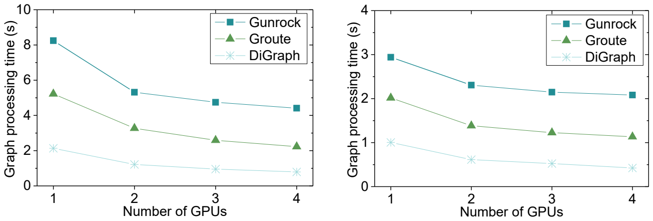


Figure 17. Total execution time of pagerank on webbase



(a) pagerank (b) SSSP

Figure 16. Comparison of scalability over webbase

average lengths of the generated paths in DiGraph are 4.2, 10.9, 3.8, 9.7, 9.4, and 3.5 for dblp, cnr, ljournal, webbase, it04, and twitter, respectively.

Figure 12 gives the traffic volume (including the volume of data transferred between GPUs and the volume of data loaded into GPU core) of pagerank. Obviously, under any circumstances, the traffic volume of DiGraph is less than both Gunrock and Groute. There are two main reasons. First, DiGraph needs much fewer vertex updates to converge than the other systems and also only needs to access a few paths for exactly once when processing a path. Second, DiGraph ensures higher utilization ratio of the loaded data than them. Figure 13 describes the ratio of the total number of used times of all vertices to the total number of loaded vertices for pagerank over Gunrock, Groute, and DiGraph, respectively. We can observe that the ratio of DiGraph is higher due to efficient graph partitioning and path-based processing/scheduling. It indicates that less time is wasted on the loading of useless data.

Finally, Figure 14 also depicts the impacts of the ratio of bi-directional edges on pagerank over different systems via adding directed edges on webbase. We observe that pagerank still gets benefits from our approach, although all edges are bi-directional ones. Note that only dependency-aware path dispatching scheme is infeasible under such condition.

4.3 Scalability over Multiple GPUs

We firstly evaluate the average GPU utilization ratios of different systems. As depicted in Figure 15, the ratios of both Groute and DiGraph are higher than Gunrock. For example, for each round of pagerank over cnr, the ratio of Gunrock is only 75.8% and 63.4% of those of Groute and DiGraph, respectively. As observed, the main reason is that Gunrock suffers from skewed load distribution of active vertices and the barriers between iterations. However, both Groute and DiGraph use the asynchronous graph processing way and have no barrier between iterations, which contributes to the higher utilization ratio of GPU. Besides, DiGraph usually enables higher utilization ratio than Groute for less data access cost.

It also means better scalability of them than Gunrock as shown in Figure 16, which gives the graph processing time of pagerank and SSSP over Gunrock, Groute, and DiGraph. For example, when the number of GPUs increases from one to four, graph processing time of pagerank is only reduced from 8.2 to 4.4 seconds over Gunrock, while its speedups over DiGraph and Groute are up to 2.7 times and 2.3 times, respectively. It also shows that the scalability of Groute is poorer than DiGraph, because Groute needs to handle much more vertex updates (more communication cost is also generated) when the number of GPUs is larger. Figure 17 also shows the total execution time of pagerank on webbase under different number of CPU threads to preprocess graph and different number of GPUs to concurrently handle graph. The results show that the preprocessing step of DiGraph also has good scalability as the others, and DiGraph still gets better performance.

5 Related Work

CPU-based Graph Processing Systems. Pregel [27], as the earliest one of typical distributed systems [9, 21, 24, 35, 38, 41, 55], expresses iterative graph algorithm as multiple iterations for simple programming. GraphLab [23], FBSGraph [52], and CoRAL [42] asynchronously execute graph algorithm without the global barrier, getting faster convergence speed and lower synchronization cost than the synchronous model used in Pregel. PowerGraph [8] evenly divides graph edges into partitions and uses a *Gather-Apply-Scatter* (GAS) model to handle vertex-programs over edges for balanced load. Lazygraph [45] proposes a lazy data coherency for replicas for lower synchronization cost. PowerSwitch [48] uses a hybrid execution mode the optimal performance. TurboGraph++ [18] uses the nested windowed streaming model for distributed graph processing with no compromising of scalability or efficiency.

Meanwhile, disk-based systems [2, 5, 10, 19, 32, 43] are also designed to support graph processing on PC for cost effectiveness. GraphChi [20] uses a parallel sliding windows method to reduce random I/O accesses. X-Stream [33] employs an edge-centric model for low data access cost via sequential access of edges. GridGraph [57], PathGraph [49], and NXgraph [6] propose two-level hierarchical partitioning scheme, tree-based partitioning scheme, and destination-sorted subshard structure for high locality, respectively. Gorder [47] tries to place vertices with common in-neighbors closer in the memory for better spatial locality. HotGraph [51] and Wonderland [50] constructs backbone structure to serve as a fast track for cross-partition state propagation for faster convergence speed. DGraph [53] is a system dedicated to disk-based directed graph processing by reducing I/O overhead, however, also suffers from ineffective vertex state propagation along directed paths and high runtime overhead. GraphGrind [39] try to reduce the impacts of graph partitioning on NUMA platform. Mosaic [26] uses Hilbert-ordered tiles to represent graph and a hybrid execution model to utilize CPU and coprocessors.

GPU-based Graph Processing Systems. Recently, several systems [7, 11, 15, 17] are proposed to exploit powerful ability of the GPU for graph processing. As one of the earliest GPU-based systems, Medusa [54] demonstrates the potential of graph processing on GPUs. For coalesced memory access and high GPU utilization ratio, CuSha [16] proposes two novel graph representations, i.e., G-shards and concatenated windows. Tigr [34] tries to translate an irregular graph into a regular representation way for efficient processing on GPU. GraphReduce [36] efficiently integrate the heterogeneous parallelism of CPU and GPU for efficient graph processing by adopting a hybrid GAS model. Garaph [25] ensures balanced vertex replication for low write contention.

On the other hand, some systems are designed for graph processing on the platform with multiple GPUs. Lux [13] utilizes the aggregate memory bandwidth across a multi-GPU cluster for efficient graph processing. GPMA+ [37] tries to efficiently support dynamic graph processing on GPUs by maximizing coalesced memory access. Gunrock [30, 46] uses a bulk-synchronous data-centric frontier-focused abstraction to naturally map graph processing to GPU. It is also extended to support efficient graph processing on multiple GPUs by using direction optimizing traversal and a just-enough memory allocation scheme. However, these synchronous methods suffer from slow vertex state propagation, imbalanced load, and under-utilization of multiple GPUs for strict synchronization between iterations. Therefore, Groute [4] is designed to use asynchronous execution model, where the computation and communication of each GPU thread can be done without waiting for other GPU threads to reach global barriers. However, it faces exceptionally high cost to repeatedly handle many vertices and also to transfer much unnecessary data.

6 Conclusion

This paper observes important characteristics of vertex update dependency in iterative directed graph processing, which enable faster convergence speed and much lower redundant data access overhead over GPUs. A path-based asynchronous execution model is proposed and several fundamental challenges are then tackled to take advantage of the observations, so as to improve the performance of GPU-accelerated iterative directed graph processing. Experimental results show that our method outperforms the state-of-the-art solutions for iterative directed graph algorithms over multiple GPUs. In the future, we will research how to reduce the communication cost between GPUs for our system via using NVLink, how to extend our approach to efficiently support the analysis of evolving directed graph on GPUs, and also extend it so as to efficiently exploit the parallelism of both CPU and GPUs for large-scale directed graph processing. In addition, we will also use our observations to guide the design of ASIC accelerator (e.g., ReRAM-based accelerator) for directed graph processing for much better performance yet with much lower energy consumption.

Acknowledgments

We would like to thank our shepherd Madan Musuvathi and all anonymous reviewers for their constructive comments and suggestions. This paper is supported by National Key Research and Development Program of China under grant No. 2018YFB1003500, National Natural Science Foundation of China under grant No. 61832006, 61825202, 61702202, and 61628204. Bingsheng's work is in part supported by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122).

References

[1] 2016. Laboratory for Web Algorithmics. Datasets. http://law.di.unimi.it/datasets.php.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 125–137.

[3] Shumeet Baluja, Rohan Seth, D. Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. 2008. Video Suggestion and Discovery for YouTube: Taking Random Walks Through the View Graph. In *Proceedings of the 17th International Conference on World Wide Web*. 895–904.

[4] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM Sigplan Symposium on Principles and Practice of Parallel Programming*. 235–248.

[5] Jiefeng Cheng, Qin Liu, and Zhenguo Li. 2015. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proceedings of the 2015 IEEE International Conference on Data Engineering*. 124–134.

[6] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An efficient graph processing system on a single machine. In *Proceedings of the 2016 IEEE International Conference on Data Engineering*. 409–420.

[7] Abdullah Gharaibeh, Lauro Beltro Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21th International Conference on Parallel Architectures and Compilation Techniques*. 345–354.

[8] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30.

[9] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. 599–613.

[10] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 246–260.

[11] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. 233–245.

[12] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining*. 538–543.

[13] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat Mccormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[14] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[15] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*. 39–50.

[16] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. 239–252.

[17] Min Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A Fast and Scalable Graph Processing Method based on Streaming Topology to GPUs. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 447–461.

[18] Seongyun Ko and Wook Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 395–410.

[19] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient Processing of Large Graphs via Input Reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 245–257.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[21] Xue Li, Mingxing Zhang, Kang Chen, and Yongwei Wu. 2018. ReGraph: A Graph Processing Framework That Alternately Shrinks and Repartitions the Graph. In *Proceedings of the 2018 International Conference on Supercomputing*. 172–183.

[22] David Liben-Nowell and Jon Kleinberg. 2007. The Link-prediction Problem for Social Networks. *Journal of the American Society for Information Science and Technology* 58, 7 (2007), 1019–1031.

[23] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[24] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.

[25] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 195–207.

[26] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems*. 527–543.

[27] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. 135–146.

[28] Ulrich Meyer. 2001. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*. 797–806.

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. *The PageRank citation ranking: Bringing*

order to the web. Technical Report. Stanford Digital Library Technologies Project.

[30] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU Graph Analytics. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium.* 479–490.

[31] Bryan Perozzi and Leman Akoglu. 2014. Focused clustering and outlier detection in large attributed graphs. In *Proceedings of the 2014 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 1346–1355.

[32] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles.* 410–424.

[33] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles.* 472–488.

[34] Amir H. N. Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-friendly Graph Processing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems.* 622–636.

[35] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* 979–990.

[36] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the 2015 International Conference for High performance Computing, Networking, Storage and Analysis.* 28:1–28:12.

[37] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 107–120.

[38] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *Proceedings of the VLDB Endowment* 12, 2 (2018), 154–168.

[39] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: addressing load imbalance of graph partitioning. In *Proceedings of the 2017 International Conference on Supercomputing.* 16:1–16:10.

[40] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[41] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An experimental comparison of partitioning strategies in distributed graph processing. *Proceedings of the VLDB Endowment* 10, 5 (2017), 493–504.

[42] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems.* 223–236.

[43] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *Proceedings of the 2016 USENIX Annual Technical Conference.* 507–522.

[44] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems.* 389–404.

[45] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. 2018. Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 276–289.

[46] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM Sigplan Symposium on Principles and Practice of Parallel Programming.* 11:1–11:12.

[47] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data.* 1813–1828.

[48] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *Proceedings of the 2015 ACM Sigplan Symposium on Principles and Practice of Parallel Programming.* 194–204.

[49] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast Iterative Graph Computation: A Path Centric Approach. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis.* 401–412.

[50] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems.* 608–621.

[51] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Guang Tan, and Bing Bing Zhou. 2017. HotGraph: Efficient Asynchronous Processing for Real-world Graphs. *IEEE Trans. Comput.* 66, 5 (2017), 799–809.

[52] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating Asynchronous Graph Processing via Forward and Backward Sweeping. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 895–907.

[53] Yu Zhang, Xiaofei Liao, Xiang Shi, Hai Jin, and Bingsheng He. 2018. Efficient Disk-based Directed Graph Processing: A Strongly Connected Component Approach. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 830–842.

[54] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.

[55] Amelie Chi Zhou, Shadi Ibrahim, and Bingsheng He. 2017. On Achieving Efficient Data Transfer for Graph Processing in Geo-Distributed Datacenters. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems.* 1397–1407.

[56] Junfeng Zhou, Shijie Zhou, Jeffrey Xu Yu, Hao Wei, Ziyang Chen, and Xian Tang. 2017. DAG Reduction: Fast Answering Reachability Queries. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data.* 375–390.

[57] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference.* 375–386.