Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation

Conference Paper · February 2018

DOI: 10.1145/3178487.3178508

7 authors, including:

Lei Wang
Chinese Academy of Sciences
22 PUBLICATIONS **280** CITATIONS
SEE PROFILE

Chen Junhang
Chinese Academy of Sciences
2 PUBLICATIONS **19** CITATIONS
SEE PROFILE

Fang Lu
Chinese Academy of Sciences
16 PUBLICATIONS **141** CITATIONS
SEE PROFILE

Xiaobing Feng
Chinese Academy of Sciences
112 PUBLICATIONS **2,110** CITATIONS
SEE PROFILE

LazyGraph: Lazy Data Coherency for Replicas in Distributed Graph-Parallel Computation

Lei Wang[1], Liangji Zhuang[1,2], Junhang Chen[1,2], Huimin Cui[1,2,*], Fang Lv[1], Ying Liu[1], Xiaobing Feng[1,2]
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
{wlei, zhuangliangji, chenjunhang, cuihm, flv, liuying2007, fxb}@ict.ac.cn

Abstract

Replicas [1] of a vertex play an important role in existing distributed graph processing systems which make a single vertex to be parallel processed by multiple machines and access remote neighbors locally without any remote access. However, replicas of vertices introduce data coherency problem. Existing distributed graph systems treat replicas of a vertex v as an atomic and indivisible vertex, and use an eager data coherency approach to guarantee replicas atomicity. In eager data coherency approach, any changes to vertex data must be immediately communicated to all replicas of v, thus leading to frequent global synchronizations and communications.

In this paper, we propose a lazy data coherency approach, called LazyAsync, which treats replicas of a vertex as independent vertices and maintains the data coherency by computations, rather than communications in existing eager approach. Our approach automatically selects some data coherency points from the graph algorithm, and maintains all replicas to share the same global view only at such points, which means the replicas are enabled to maintain different local views between any two adjacent data coherency points. Based on PowerGraph, we develop a distributed graph processing system LazyGraph to implement the LazyAsync approach and exploit graph-aware optimizations. On a 48-node EC2-like cluster, LazyGraph outperforms PowerGraph on four widely used graph algorithms across a variety of real-world graphs, with a speedup ranging from 1.25x to 10.69x.

CCS Concepts **•Computing methodologies → Distributed programming languages; Parallel programming languages**

Keywords Lazy Data Coherency, Distributed Graph-parallel Computation, Execution Model

ACM Reference format:
Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, Xiaobing Feng. 2018. LazyGraph: Lazy Data Coherency for Replicas in Distributed Graph-Parallel Computation. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18). Vienna, Austria,* 13 pages.
https://doi.org/10.1145/3178487.3178508

1 Introduction

Efficient processing of large-scale graphs has gain significant interest in both academia and industry recently. Due to the desire to process tremendous graphs, many graph processing systems have been proposed and been able to run on distributed machines [1-14].

Replicas of vertices play an important role in existing distributed graph processing systems. When placing a graph-structured data across multiple machines, a single vertex is spanned to multiple machines. Thus, replicas v_0, v_1, ..., v_k of a vertex v make the single vertex v to be parallel processed by multiple machines, thus each vertex can access remote neighbors locally via the corresponding replicas without any remote access. Many graph partitioning algorithms [3,4,8,33,20-25] are proposed to minimize the communication cost through reducing the number of replicas.

However, replicas of vertices introduce the data coherency problem. Existing distributed graph systems treat all the replicas v_0, v_1, ..., v_k of a vertex v as an atomic and indivisible vertex, and use an **eager data coherency** approach to guarantee the atomicity. In the eager data coherency approach, any changes to the vertex v must be immediately communicated to all of its replicas, thus the communication overhead is determined by the number of machines spanned by each vertex and the frequency of data synchronization for each vertex. The atomicity is strictly maintained no matter in asynchronous or synchronous engines. In particular, changes to vertex data are copied to all replicas of v as soon as possible in the asynchronous engine, while these changes are batch-processed in the synchronous engine. Therefore, the eager data coherency approach leads to frequent global synchronizations and communications between replicas of a vertex, and introduces significant overhead.

[*] To whom correspondence should be addressed.

PPoPP '18, *February 24–28, 2018, Vienna, Austria*

https://doi.org/10.1145/3178487.3178508

For most graph algorithms, the data coherency between replicas of a vertex can be delayed. Let us consider two scenarios. First, in some graph algorithms, such as k-core, breadth-first search and connected components, the solution of a vertex depends only on a subset of its neighbors. A replica can update its own vertex data according to its local messages and send new messages to its local neighbors. But in eager data coherency approach, a replica has to wait for all replicas to finish their message collection, thus introducing unnecessary waiting cost. Second, for algorithms such as PageRank [49] and loopy belief propagation [50], an iterative equation can be expressed as $x_i^{(t+1)} = x_i^{(t)} +_{op} \oplus_{j \to i \in E} \Delta_j^{(t)}$, where i is an vertex id, t is an iterative count, $+_{op}$ and $\oplus$ represent operations, $x_i^{(t)}$ is vertex data of i at iteration t, $\Delta_j^{(t)}$ is a delta value (a change value) of vertex j at iteration t. The vertex-program receives messages $\Delta_j^{(t)}$ of all in-neighbors, and then updates the new $x_i^{(t+1)}$ based on $x_i^{(t)}$, and sends $\Delta_i^{(t+1)}$ to its out-neighbors. The solution of a vertex can incrementally change from the initial value, until a convergence condition is reached. So a replica can receive messages of multiple iterations from its local neighbors and then communicate to other replicas.

In this paper, we propose a **lazy data coherency** approach, called *LazyAsync*, to asynchronously execute replicas of a vertex and reduce the number of global synchronization and network traffic. The key idea is that replicas of a vertex are treated as independent vertices, and the coherency between these replicas are maintained via calculation only at the data coherency points, which are automatically selected from the algorithm by our approach. *LazyAsync* works in the following steps. First, some edges are heuristically identified to be split into *parallel-edges*. In particular, an edge $v \to u$ is split into max(|V|,|U|) *parallel-edges*, where V and U represent the replicas for *v* and *u* respectively. Second, the graph is being processed on all machines in the lazy mode between two adjacent coherency points. Specifically, each replica maintains a version of the vertex data and updates its vertex data using $x_i^{(t+1)} = x_i^{(t)} +_{op} \oplus_{j \to i \in E} \Delta_j^{(t)}$. During the processing, since V and U are located on the same machine, the message passing along the *parallel-edges* can be implemented as local write operations, without introducing any global synchronizations or communications. Finally, when the graph algorithm reaches the data coherency point, each machine collects the messages (delta) from all replicas, aggregates the messages (delta) together and updates the vertex data via some computations to obtain the global view. Note that for graph algorithms, the vertex data can be determined only by the initial value and the messages, in regardless of the order of these messages.

Based on PowerGraph, we develop a distributed graph processing system LazyGraph to implement the *LazyAsync* approach. LazyGraph consists of two parts: First, for programmers, LazyGraph maintains the same programming interface of existing distributed graph processing systems, but it requires the programmers to write the graph algorithms into push-style vertex-programs and delta propagations. Second, LazyGraph runtime provides the underlying support for *LazyAsync*, including *one-edge* and *parallel-edges* message transmission modes, together with a mechanism to maintain the coherency of the replicas, by expressing the computation of $v_0, v_1, \ldots, v_k$ distributed on different machines as a sequence of *local computation* stages and *data coherency* stages.

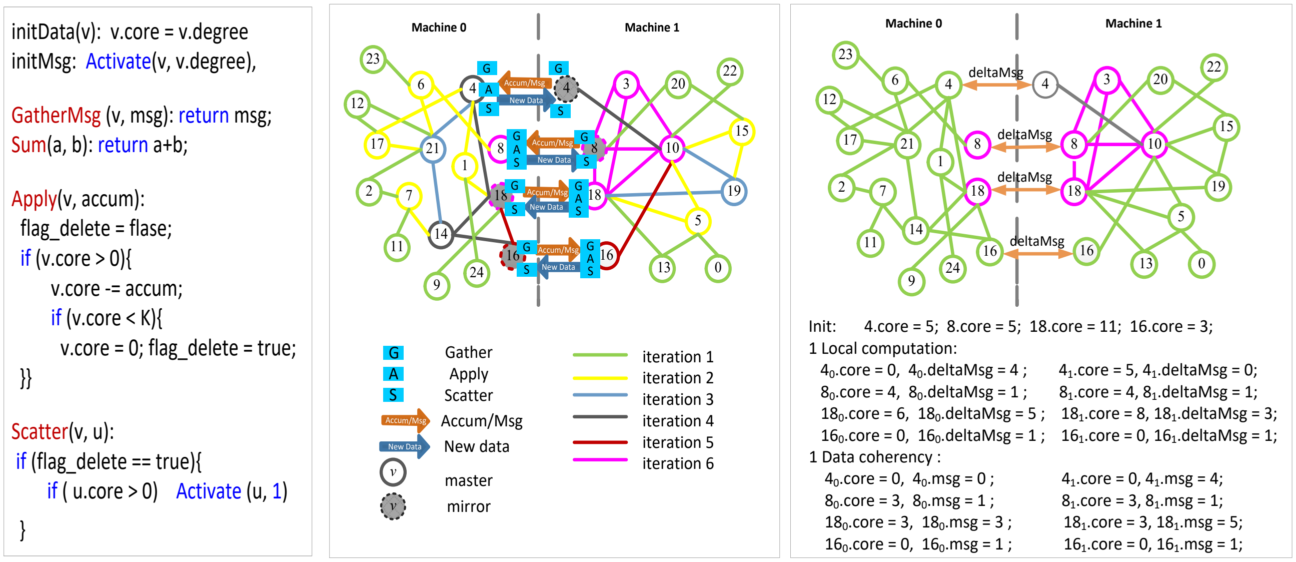
This paper makes the following contributions:

1) We propose a lazy data coherency approach *LazyAsync* for distributed graph processing. It automatically selects some data coherency points from the graph algorithm, and maintains the replicas to share the same global view only at such points, which means the replicas of a vertex are enabled to maintain different local views between any two adjacent data coherency points, and thus reduces the number of global synchronization and communication volume, and substantially accelerates the convergence of graph algorithms.
2) We propose a *parallel-edges* message transmission mechanism, which converts a message transmission between replicas of two different vertices into multiple local write operations, to avoid remote messages and achieve rapid convergence on local computation.
3) Based on PowerGraph, we develop a distributed graph processing system LazyGraph to implement *LazyAsync* approach and exploit the graph-aware optimizations. LazyGraph contains three graph-aware optimizations including graph partitioning to support *one-edge* and *parallel-edges* message transmission modes, adaptive interval strategy selection between two adjacent data coherency points, and dynamic switching between two communication modes at data coherency points, i.e., all-to-all and mirrors-to-master.
4) Our experimental results show that on a 48-node EC2-like cluster, LazyGraph outperforms PowerGraph on four widely used graph algorithms across a variety of real-world graphs, with a speedup ranging from 1.25x to 10.69x.

2 Background and Motivation

2.1 Vertex Computation and GAS Model

Distributed graph processing systems, based on the "think like a vertex" programming model [53], iteratively execute a user-defined vertex-program over vertices of a graph. Most of these systems abstract the vertex computation into the Gather-Apply-Scatter (GAS) model [3]. The Gather and Sum function collect information about the neighbors of the vertex along in-edges, like a commutative associative



```
initData(v):  v.core = v.degree
initMsg:  Activate(v, v.degree),

GatherMsg (v, msg): return msg;
Sum(a, b): return a+b;

Apply(v, accum):
 flag_delete = flase;
 if (v.core > 0){
      v.core -= accum;
      if (v.core < K){
        v.core = 0; flag_delete = true;
 }}

Scatter(v, u):
 if (flag_delete == true){
     if ( u.core > 0)   Activate (u, 1)
 }
```

(a) Pseudocode for *k-core*

(b) 3-core decomposition using Sync

```
Init:     4.core = 5;  8.core = 5;  18.core = 11;  16.core = 3;
1 Local computation:
  4_0.core = 0,  4_0.deltaMsg = 4 ;      4_1.core = 5, 4_1.deltaMsg = 0;
  8_0.core = 4,  8_0.deltaMsg = 1 ;      8_1.core = 4, 8_1.deltaMsg = 1;
  18_0.core = 6,  18_0.deltaMsg = 5 ;    18_1.core = 8, 18_1.deltaMsg = 3;
  16_0.core = 0,  16_0.deltaMsg = 1 ;    16_1.core = 0, 16_1.deltaMsg = 1;
1 Data coherency :
   4_0.core = 0,  4_0.msg = 0 ;          4_1.core = 0, 4_1.msg = 4;
   8_0.core = 3,  8_0.msg = 1 ;          8_1.core = 3, 8_1.msg = 1;
   18_0.core = 3,  18_0.msg = 3 ;        18_1.core = 3, 18_1.msg = 5;
   16_0.core = 0,  16_0.msg = 1 ;        16_1.core = 0, 16_1.msg = 1;
```

(c) 3-core decomposition using LazyAsync

Figure 1. Illustration of the difference between eager/lazy data coherency for replicas

combiner. The result is used in the Apply function, which calculates and updates the vertex value. The Scatter function sends messages and activates neighbors along out-edges.

For example, Fig.1(a) shows *k-core Decomposition* (*k-core*) algorithm pseudocode. A *k*-core of a graph is a maximal connected subgraph in which each vertex is connected to at least *k* vertices. *K-core Decomposition* is based on following iterative equations:

$$core_i^{(t+1)} = core_i^{(t)} - \sum_{j\to i \ or \ i\to j} \Delta_j^{(t)} \qquad (1)$$

$$\Delta_i^{(t+1)} = 1 \qquad if\ core_i^{(t+1)} < K \qquad (2)$$

Where t is the t iteration, $core_i$ is the value of vertex i, and Δ_i is whether vertex i is deleted or not. In Fig.1(a) an active vertex v receives messages from its neighbors by GatherMsg function, and combines these messages by Sum function. *accum* is the number of edges which v needs to delete and is returned by the GatherMsg function. In Apply function, *v.core* is updated by *accum*; if *v.core* is less than K, v is deleted. In Scatter function, if v is deleted, v sends 1 to its neighbors.

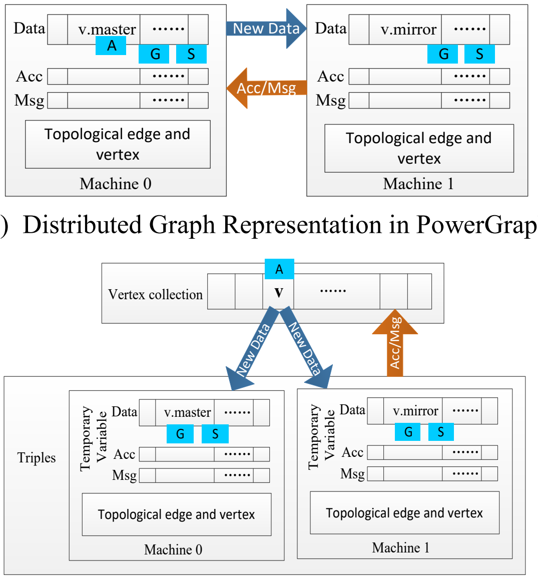
2.2 Replicas of Vertices

In distributed processing, a graph is a whole graph from a user view and is a partitioned graph from system view.

Replicas (mirror vertices): Replicas of vertices play an important role in existing distributed graph processing systems. When placing graph-structured data across multiple machines, existing distributed graph systems will generate a large number of replicas of vertices. For example, vertex-cut partitioning algorithms [3,5,8,20,21] evenly assign the edges of a graph to machines and allow the vertices to span machines, thus a single vertex can be parallelized by multiple machines and access remote neighbors locally without any remote access. Another example is edge-cut partitioning algorithms [1,2,4,6,22-25] which assign each machine a disjoint set of vertices and all the connected incoming/outcoming edges. Two vertices connected by each cut edge have two copies on both machines, thus vertices can access remote neighbors locally without any remote access. As vertex-cut partitioning achieves better load balance on power law graphs than edge-cut partitioning does, recent distributed graph systems [3,5,8,9] use vertex-cut to place graph-structured data across multiple machines. These systems will set up replicas of vertices to enable computation, and the automatic synchronization of these replicas requires communication.

Replicas of vertices introduce three issues: the data coherency between replicas of a vertex, the message transmission between replicas of two different vertices, and the order of replicas computation.

ISSUE I **(eager data coherency approach for replicas):** PowerGraph[3] and GraphX[5] represent two different implementations. Fig.2.(a)(b) illustrate the physical graph representations and implement the eager data coherency approaches in PowerGraph and GraphX respectively. To guarantee atomic update of replicas, PowerGraph allows one of replicas (the master) maintain an actual vertex data and all remaining replicas (mirrors) maintain a local cached read only copy of the vertex data. Any change to the vertex data must be made to the master which is then immediately replicated to all mirrors. GraphX



(a) Distributed Graph Representation in PowerGraph

(b) Distributed Graph Representation in GraphX

Figure 2. Eager Data Coherency for Replicas of *v*

[5] is an embedded graph processing framework built on top of Spark general dataflow framework. The automatic data coherency of these replicas is analogous to the cache coherency in shared memory multi-processors system. GraphX allows vertex collection to maintain actual vertex data and all replicas to maintain a local cached read only copy of the vertex data. When the eager data coherency approach implements the Gather-Apply-Scatter model, each phase has a global sync, each mirror sends one accumulator to the master in gather phase, and the master sends the updated vertex data to all mirrors in apply phase. The approach needs two communications and three synchronizations to update vertex data.

ISSUE II (the message transmission between replicas of two different vertices**):** When *v* sends a message *msg* to *u* along an edge $v \to u$, the systems need two communications and three synchronizations to complete this sending action. On a user view graph, *v* sends a message to *u* along an edge $v \to u$. Now on a partitioned graph, this message transmission becomes that replicas (v_0, v_1, ..., v_k) send a message to replicas (u_0, u_1, ..., u_p) on the partitioned graph. Existing distributed graph processing system uses a one-edge message transmission mode: assuming the edge $v \to u$ assigned to machine *i*, v_i collects messages from v_0, v_1, ..., v_k across the network, and sends these messages to u_j along the local edge v_i->u_j, and then u_j sends these messages to u_0, u_1, ..., u_p across the network. **This edge v_i->u_j becomes the bottleneck.**

ISSUE III (**Sync and Async engines):** A synchronous engine (Sync) and an asynchronous engine (Async) [2,3] provide different visibility timing of updated vertex data for neighbor vertices, but both engines use the eager data coherency approach to synchronize replicas of an active vertex, and use the one-edge message transmission mode. The major difference between Sync and Async modes is the order of vertices computation, which provides different visibility timing of updated vertex data for subsequent vertex computation. Changes to vertex data are copied to all replicas of *v* as soon as possible in Async engine, but these changes are batch processed in Sync engine.

2.3 Redundancy

The eager data coherency approach for replicas of a vertex introduces considerable redundancies. We use *k-core Decomposition* algorithm to illustrate this problem. For example, in Fig.1(b), the vertex-cut partitioning places a small graph (25 vertexes and 41 edges) on two machines, and let vertex 4, 8, 16, and 18 to span machines. When 3-core decomposition uses Sync engine, after 6 iterations with 12 communications and 18 global synchronizations, vertex *3, 8, 10, 18* are found to connect a 3-core subgraph. There are redundant synchronizations and communications. a) **Redundant waiting**. The local messages of 4_0 are enough to get the solution, but 4_0 has to wait the remote messages of 4_1. Similarly, the local messages of 6_0 are enough to get the solution, but 6_0 has to wait the remote messages of 6_1. b) **Frequent global synchronization and communication**. For vertex *18*, there are 15 synchronizations and 10 communications between 18_0 and 18_1. v.core of *18* is updated to 9, 7, 6, 4, 3 from iteration 2~6 respectively. c) **Redundant synchronizations**. Local vertices do not need to wait remote local vertices to complete their calculations, e.g. vertex *2*, *9*, *11*, *12*, *23*, *24* on machine 0 do not need to wait vertex *0,13, 20, 22* on machine 1 to complete their calculations at the first iteration. Although the asynchronous engine can eliminate the third redundancy, it cannot solve the first and the second redundancy.

2.4 Our approach

In this paper, we focus on leveraging the lazy data coherency for replicas of a vertex to reduce the number of global synchronizations and network traffic. We treat replicas v_0, v_1, ..., v_k of a vertex *v* as independent vertices, with each replica maintains a version of the vertex data and updates its vertex data by $x_i^{(t+1)} = x_i^{(t)} +_{op} \oplus_{j \to i \in E} \Delta_j^{(t)}$ iteration equation. Although replicas of *v* receive messages in different order, they can obtain the same value as long as they have the same initial value and receive the same messages. LazyGraph expresses the computation of replicas v_0, v_1, ..., v_k as a sequence of *local computation* stages and *data coherency* stages, and maintains the coherency between these replicas via calculation, which merge local values and remote delta messages, only at the data coherency points. Thus *LazyAsync* allows replicas of a vertex to maintain different local views of the vertex

```
initData(v):  v.rank = 0.15;  v.Δ = -0.85;
initMsg:  Activate(u, 1/v.outDegree), v->u ∈ E

GatherMsg(v, msg): return msg;
Sum(a, b): return a+b
Inverse(accum, a): return accum-a;

Apply(v, accum):
  Δ= 0.85 * accum;
  v.rank += Δ;
  Δ += v.Δ;
  if( Δ > tol ) { v.Δ= 0.0; }
  else { v.Δ = Δ;}

Scatter(v, u)
  if( Δ > tol )
    Activate(u, Δ/v.outDegree)
```

Figure 3. Pseudocode for PageRank

between any two adjacent data coherency points and share the same global view only at a data coherency point.

For example, when 3-core decomposition uses *LazyAsync* in Fig.1(c), there is only one communication and one synchronization. In *LazyAsync* 18_0 and 18_1 are independent vertices. In the beginning, 18_0.core = 11 and 18_0.deltaMsg = 0, and 18_1.core = 11 and 18_1.deltaMsg = 0. After the first local computation stage, 18_0.core = 6 and 18_0.deltaMsg = 5, and 18_1.core = 8 and 18_1.deltaMsg = 3. And then at the first data coherency stage, replicas update their local value by remote delta messages, that is 18_0.core=6–3=3 and 18_1.core=8 – 5=3.

3 Lazy Data Coherency Approach

To address the challenges of the eager data coherency for replicas of a vertex, we introduce the lazy data coherency approach, called *LazyAsync*.

1) It automatically selects some data coherency points from the graph algorithm, and maintains the replicas to share the same global view only at such points, which means the replicas of a vertex are enabled to maintain different local views between any two adjacent data coherency points. (Section 3.2).
2) It supports two message transmission modes (*one-edge* and *parallel-edges*) at the same time, but an edge of a graph only uses one of the two modes, different edges can use different modes. Thus this approach can benefit from the two modes, *one-edge* transmission mode to save memory, and *parallel-edges* mode to save transmission cost and achieves rapid convergence on local computation (Section 3.3).
3) It uses a vertex-cut and edge-split graph partitioning, which uses a vertex-cut partitioning to place graph-structured data across multiple machines, and then splits some edges into parallel edges and assigns these parallel edges to each machine (Section 4.1).
4) It provides *LazyBlockAsync* and *LazyVertexAsync* engines to schedule the order of vertex computation and to provide different visibility timing of updated vertex data for subsequent vertex computation (Section 3.4).

3.1 Vertex Computation and Applications

LazyAsync maintains the same programming interface of existing distributed graph processing systems, but it requires the programmers to write the graph algorithms into push-style vertex-programs and delta propagations. Like existing distributed graph processing systems, a user-defined vertex-program *P* in *LazyAsync* runs on a directed graph G = {V, E} and computes in parallel on each vertex $v \in$ V. Vertices communicate directly with each other by sending messages. A vertex can receive messages sent to it, modify its data, and send messages to other vertices. An edge is used to transmit messages. These actions are expressed by the GatherMsg, Sum, Inverse, Apply and Scatter functions of a vertex-program *P*. *LazyAsync* still conforms to the GAS model.

Unlike existing distributed graph processing systems, *LazyAsync* requires the vertex computation to be based on $x_i^{(t+1)} = x_i^{(t)} +_{op} \oplus_{j\to i\in E} \Delta_j^{(t)}$iterative equation, where *i* is an vertex id, *t* is an iterative count, $+_{op}$ and $\oplus$ represent operations, $\Delta_j^{(t)}$is a delta value (a change value) of vertex *j* at iteration *t*. The user defined Sum $\oplus$ operation must be commutative and associative. In gather messages phase, a vertex *i* receives messages $\Delta_j^{(t)}$ sent to it and uses the Sum $\oplus$ operation to combine these messages to *accum*. *accum* is used in apply phase to update the value of the central vertex using $x_i^{(t+1)} = x_i^{(t)} +_{op} accum$. Finally, in Scatter phase the new change (delta) $\Delta_i^{(t+1)}$ is sent to its neighbors along adjacent edges.

In Fig.3 we implement PageRank using the LazyGraph abstraction. For Single Source Shortest Path algorithms (SSSP), Connected Components (CC) and *k*-Core, *LazyAsync* and PowerGraph have the same codes. But for PageRank, *LazyAsync* uses a variant of PageRank (PageRank-Delta). PageRank is widely used to evaluate the relative importance of webpages; the rank of webpages is based on Equation (3).

$$\mathrm{PR}(i) = 0.15 + 0.85 \sum_{j\to i\in E} \frac{PR(j)}{\mathrm{outDeg}(j)} \tag{3}$$

Here we use PageRank-Delta in which $PR_i^{(0)}$ and $PR_i^{(1)}$ are initial values, and $PR_i^{(t+1)}$applies the iterative Equation (4).

$$PR_i^{(t+1)} = PR_i^{(t)} + 0.85 \sum_{j\to i\in E} \frac{PR_j^{(t)} - PR_j^{(t-1)}}{\mathrm{outDeg}(j)} \tag{4}$$

In PageRank-Delta, the GatherMsg and Sum functions receive the total delta contributions from neighbours, the

Apply function computes the new PR value and accumulates the delta value, the Scatter function sends the change (delta) in PR value to neighbours if it has accumulated enough change (delta) in its PR value. In our implementation, each vertex is initialized to $PR_i^{(init)} = 0.15$ and $\Delta_i^{(init)} = -0.85$, and each edge is initialized to $msg_{j\rightarrow i\in E}^{(init)} = \frac{1}{\text{outDeg}(j)}$. And then after the first iteration, we can obtain $PR_i^{(1)} = 0.15 + 0.85\sum_{j\rightarrow i\in E}\frac{1}{\text{outDeg}(j)}$ and $\Delta_i^{(1)} = PR_i^{(1)} - PR_i^{(0)} = -0.85 + 0.85\sum_{j\rightarrow i\in E}\frac{1}{\text{outDeg}(j)}$.

3.2 Lazy Data Coherency for Replicas

In *LazyAsync*, a user-defined vertex-program *P* runs on a directed graph G = {V, E} in user view, but it actually runs on a partitioned graph G in runtime system. When *LazyAsync* places graph G across multiple machines, it splits a vertex into multiple replicas distributed on different machines. The runtime system will translate the API functions of a user-defined vertex-program *P* into low level graph operators, and use *LazyAsync* to asynchronously parallelize the computation of a single vertex.

We now show how to use the lazy data coherency for replicas to asynchronously parallelize the computation of a single vertex. *LazyAsync* redefine how to calculate the vertex data of replicas v_0, v_1, ..., v_k in the runtime system and express the computation of replicas v_0, v_1, ..., v_k distributed on different machines as a sequence of *local computation* stages and *data coherency* stages (shown in Fig.4). In the *local computation* stage, replicas of a vertex maintain different local views of this vertex, and update their local views with received messages from local neighbors on the same machine; the new local views can be visible to local neighbors immediately without waiting a data coherency point; and replicas accumulate delta messages received from local neighbors along the *one-edge* transmission mode during a local computation stage. In the *data coherency* stage, each replica sends its delta message to the other remote replicas across the network and receives delta messages of all the other remote replicas; and then replicas run the *apply* operator on their local views with others delta messages as parameter, and thus obtain the same global view.

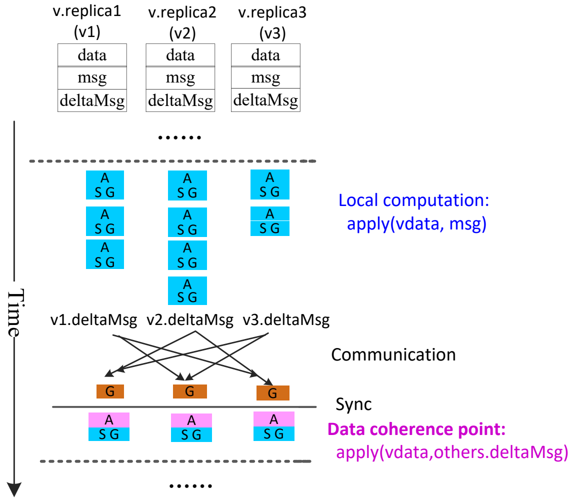
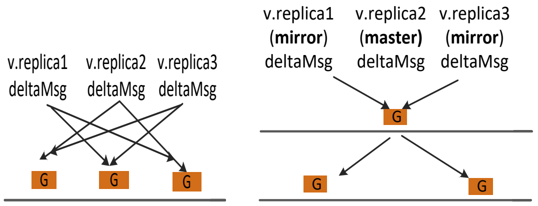


Figure 4. Lazy Data Coherency for replicas of *v*



(a) all-to-all mode (b) mirror-to-master mode

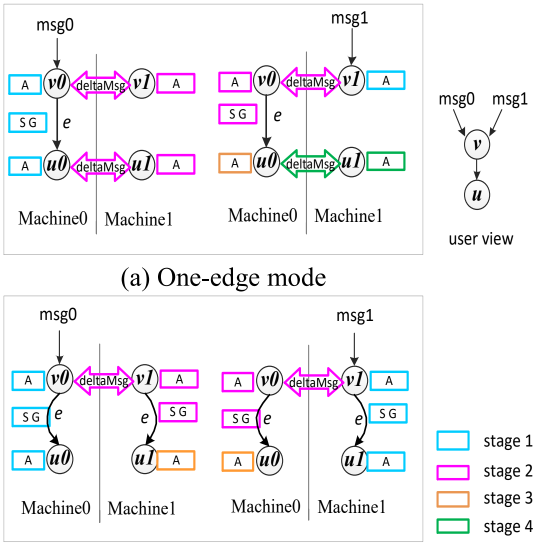
Figure 5. Two communication modes

For each replica *v* on each machine, the runtime system main-tains a number of variables:

- *vdata*[*v*], the local vertex data
- *message*[*v*], the message sent to *v* from its neighbors
- *deltaMsg*[*v*], the delta message accumulating all messages which *v* receives from local neighbors along the *one-edge* transmission mode during a local computation stage
- *replicas*[*v*], the machines on which the replicas of *v* are placed
- *isActive*[*v*], a state identifying *v* active or inactive

The runtime system translates the API functions of a user-defined vertex-program into a number of low level graph operators, and uses LazyAsync to asynchronously parallelize the computation of a single vertex.

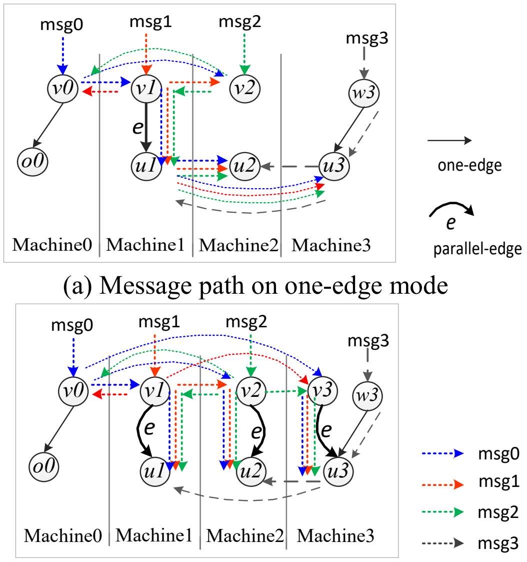
- Init(*v*), this operator uses the user initData and initMsg functions to initialize vertices and messages
- Apply(*v*, *message*[*v*]), this operator uses the user Apply function to update *vdata*[*v*], and then clear *message*[*v*]
- ScatterGatherMsg(*v, u*), when *v* sends a message *msg* to local neighbor *u* along a local edge, this operator directly writes this message *msg* into *message*[*u*] using the user Scatte, GatherMsg and Sum functions. If the edge uses *one-edge* transmission mode, this message *msg* is accumulated into *deltaMsg*[*u*]; if the edge uses *parallel-edge* transmission mode, *msg* isn't accumulated into *deltaMsg*[*u*]. Finally, this operator sets *u* to active and sets *v* to inactive.
- Exchange_deltaMsg(*v, deltaMsg*[*v*]), this operator has all-to-all and mirrors-to-master communication modes (shown in Fig.5), and allows dynamically switching between the two modes to gain optimal performance. In all-to-all mode (shown in Fig.5(a)), a replica *v* sends its *deltaMsg*[*v*] to the other remote replicas of this vertex across the network, and receives delta messages of all the other remote replicas; accumulates other delta messages into *message*[*v*] using the user Sum function,



(a) One-edge mode

(b) Parallel-edges mode

Figure 6. Two message transmission modes, *v* sends messages to *u* along one edge or along parallel-edges.



(a) Message path on one-edge mode

(b) Message path on two transmission modes

Figure 7. Message transmission path

and clears *deltaMsg*[*v*]. In mirrors-to-master mode (shown in Fig.5(b)), all mirrors send their *deltaMsg* to master across the network, and then master combines all delta messages using the Sum function and sends the new delta message to all mirrors; each replica subtracts its *deltaMsg*[*v*] from the new delta message using the Inverse function, accumulate the new delta message into *message*[*v*] by using the Sum function, and clear *deltaMsg*[*v*].

3.3 Two Message Transmission Modes

LazyAsync supports two message transmission modes, message transmission along an edge or along parallel-edges of an edge. When *v* sends messages to *u*, if the edge *v* → *u* uses *one-edge* mode, these messages will reach replicas of *u* by the data coherency between replicas of *u*; if the edge *v* → *u* uses *parallel-edges* mode, these messages will reach replicas of *u* along the local edge *v* → *u*. Specifically, in the ***one-edge* transmission mode** (shown in Fig.6(a)), an edge *v* → *u* is assigned to a machine and connects a pair of replica v_i and replica u_j. When replicas of *v* send messages to all replicas of *u*, firstly each replica *v* sends its delta message to the other remote replicas across the network, and all replicas of *v* obtain the same global view through the data coherency; and then replica v_i sends a message to replica u_j along the edge *v->u*; finally u_j sends this delta message to all other replicas of *u* at the data coherency stage. In the ***parallel-edges* transmission mode** (shown in Fig.6(b)), an edge *v->u* is split into multiple parallel-edges, and each parallel-edges connects each pair of replica v_i and replica u_i. When replicas of *v* send messages to all replicas of *u*, firstly each replica *v* sends its delta message to other remote replicas across the network, and all replicas of *v* obtain the same global view through the data coherency; and then each replica *v* send a message to each replica *u* along a local edge of parallel-edges. There is no the data coherency among replicas of *u*, because messages from all replicas of *v* arrive each replica *u* via the local edge of parallel-edges. The parallel-edges mode is only suitable for the edge with read-only values.

Fig.6 shows the two message transmission modes, in a user view, a vertex *v* receives messages *msg0* and *msg1*, updates its data, and then sends a message to its neighbor *u* along an edge *v->u*. Fig.6(a) is a one-edge mode, an edge *v->u* is assigned to machine 0, a replica of *v* (*v0*) receives *msg0*, another replica *v1* receives *msg1*. Propagating *msg0* to *u* needs two stages, and propagating *msg1* to *u* needs four stages. Fig.6(b) is a parallel-edges mode, the parallel-edges of an edge *v->u* are assigned on two machines, *v0* receives *msg0*, *v1* receives *msg1*. Propagating *msg0 or msg1* to *u* needs three stages. In the parallel-edges mode, edges are not the bottleneck, and messages can be sent to neighbours as soon as possible.

Our distributed graph system supports the two message transmission modes at the same time, but an edge of a graph only uses one of the two modes, different edges can use different modes. Some edges of a vertex *u* may use the one-edge transmission mode, and some other edges may use the parallel-edges mode (shown in Fig.7(b)).

How does this vertex *u* maintain the data coherency of replicas? When a replica u_i receives a message from its neighbors along a local edge, if this local edge is the one-edge transmission mode, this message is accumulated into the delta message of u_i; otherwise this message isn't accumulated into the delta message of u_i. Thus at the data

Algorithm 1. LazyBlockAsync Engine

```
Input: G(V, E, D)
Input: Initial active vertex set activeCurr
while(iteration <= max_iteration) {

Stage1: local computation stage
  if(doLocal){
    parallel_for( acitveCurr ){
      if (activeCurr == NULL || !doLC())  break;
      Applys();
      ScatterGatherMsgs();
      activeCurr = activeNext; activeNext = NULL;
        }}

      Stage2: data coherency stage
        Exchange_deltaMsgs();
        barrier();
        if(msgEmpty() && activeEmpty())   break;
        if(!doLocal && turnOnLazy() )
          doLocal = 1;
        // data coherency point
        parallel_for( acitveCurr ){
          Applys();
          ScattersGatherMsgs();
          acitveCurr = NULL;
        }

        activeCurr = activeNext; activeNext = NULL;
        iteration ++;
}
```

coherency stage, only messages from the one-edge transmission mode are exchanged, and each replica *u* updates its data.

Fig.7 shows message transmission path on the two transmission modes. In Fig.7(a) an edge *v->u* is the one-edge transmission mode and is assigned in machine 1. *v* is split into multiple replicas *v0*, *v1*, and *v2* in machine 0, 1 and 2. *u* is split into multiple replicas *u1*, *u2*, and *u3* in machine 1, 2 and 3. Messages received by *v0* and *v2* are sent to *v1*, and then reach *u1* along the edge *v->u*, and finally are sent to *u2* and u3 by *u1*. In Fig.7(b) an edge *v->u* uses parallel-edges transmission mode and an edge *w->u* uses one-edge transmission mode. The edge *v->u* must be assigned to all the machines replicas of *u* stored on. Messages received by *v0, v1* and *v2* are sent to *v1, v2* and *v3,* and then reach *u1*, *u2*, and *u3* along parallel-edges *v1->u1*, *v2->u2* and *v3->u3,* respectively. Message *msg3* received by *w3* reaches *u3* along the edge *w->u* and then are sent to *u1* and *u2*.

3.4 LazyBlockAsync and LazyVertexAsync Engines

LazyAsync provides *LazyBlockAsync* and *LazyVertexAsync* engines to schedule the order of vertex computation and to provide different visibility timing of updated vertex data for subsequent vertex computation. Algorithm 1 shows the *LazyBlockAsync* engine. All the vertices enter local computation stage and data coherency stage at the same time, and a global barrier to synchronize vertex execution follows the delta messages exchanges in the data coherency stage. It is possible for batched data update and well-optimized network message dispatching with high resource utilization. Algorithm 2 shows the *LazyVertexAsync* engine, which has no global barrier to synchronize vertex execution, and the updated vertex global view is visible to neighboring vertices as soon as possible. The *LazyVertexAsync* engine emphasizes the fast convergence speed, and hides the network latency by pipeline of vertex processing.

3.5 Correctness

In this section, we state the main correctness results of the lazy data coherency approach *LazyAsync*.

PROOF. Firstly, given the initial values $x_i^{(0)}$and $x_i^{(1)}$, the new value x_i of the vertex *i* applies the iterative equation $x_i^{(t+1)} = x_i^{(t)} +_{op} \bigoplus_{j \to i \in E} \Delta_j^{(t)}$. Since the sum ⊕ operation defined by the user must be commutative and associative, the value accum $= \bigoplus_{j \to i \in E} \Delta_j^{(t)}$ is independent of the sequence of messages $\Delta_j^{(t)}$ along an edge $j \to i$.

The iterative equation can be expressed as in

$$x_i^{(t+1)} = x_i^{(t)} +_{op} \Delta_{j1}^{(t)} +_{op} \Delta_{j2}^{(t)} +_{op} \cdots +_{op} \Delta_{jk}^{(t)}; \qquad j_{1\ldots k} \to i \in E$$

and the new value $x_i^{(t+1)}$ is also independent of the sequence of messages $\Delta_{j_{1\ldots k}}^{(t)}$.

And then the iterative equation can be expanded into multiple iterations as in:

$$x_i^{(t+1)} = (x_i^{(t-1)} +_{op} \bigoplus_{j \to i \in E} \Delta_j^{(t-1)}) +_{op} \bigoplus_{j \to i \in E} \Delta_j^{(t)}$$
$$= x_i^{(t-1)} +_{op} \Delta_{j1}^{(t-1)} +_{op} \cdots +_{op} \Delta_{jk}^{(t-1)} +_{op} \Delta_{j1}^{(t)} +_{op} \cdots +_{op} \Delta_{jk}^{(t)}$$
$$= x_i^{(1)} +_{op} \bigoplus_{j \to i \in E} \bigoplus^{iter \in 1 \sim t} \Delta_j^{(iter)}; \quad j_{1\ldots k} \to i \in E$$

and the new value $x_i^{(t+1)}$ is obtained by $x_i^{(1)}$ accumulating these delta messages $\Delta_{j_{1\ldots k}}^{(t_{1\ldots t})}$, and is also independent of the

Algorithm 2. LazyVertexAsync Engine

```
Input: G(V, E, D)
Input: Initial active vertex set activeCurr
while(!activeVEmpty()) {
   v = dequeue(acitveCurr);
   if (!needDataCoherency(v)) {
Stage1: local computation stage
      Applys();
      ScatterGatherMsgs();
      enqueue(activeCurr);
   } else {
Stage2: data coherency stage
            Exchange_deltaMsgs();
            // data coherency point
            Applys();
            ScattersGatherMsgs();
            enqueue(activeCurr);
         }
      }
```

sequence of these delta messages.

Secondly, assuming x_i is split into three replicas x_{i1}, x_{i2} and x_{i3}, the edges $j_{1 \ldots k} \rightarrow i$ are equally assigned to each replica of x_i, e.g. the edges $j_{1 \ldots p} \rightarrow i1$, the edges $j_{p+1 \ldots l} \rightarrow i2$, and the edges $j_{l+1 \ldots k} \rightarrow i3$. These edges use *one-edge* mode. The three replicas respectively apply these iterative equations as in:

$$x_{i1}^{(t+1)} = x_{i1}^{(t)} +_{op} \oplus_{local\ j_{1 \ldots p} \rightarrow i1} \Delta_j^{(t)} +_{op} \oplus_{remote\ j_{p+1 \ldots k} \rightarrow i2,3} \Delta_j^{(t)}$$

$$x_{i2}^{(t+1)} = x_{i2}^{(t)} +_{op} \oplus_{local\ j_{p+1 \ldots l} \rightarrow i2} \Delta_j^{(t)} +_{op} \oplus_{remote\ j_{1 \ldots p, l+1 \ldots k} \rightarrow i1,3} \Delta_j^{(t)}$$

$$x_{i3}^{(t+1)} = x_{i3}^{(t)} +_{op} \oplus_{local\ j_{l+1 \ldots p} \rightarrow i3} \Delta_j^{(t)} +_{op} \oplus_{remote\ j_{1 \ldots l} \rightarrow i2,3} \Delta_j^{(t)}$$

And then the three replicas respectively apply these iterative equations as in:

$$x_{i1,2,3}^{(t+1)} = x_{i11,2,3}^{(1)} +_{op} \oplus_{j \rightarrow i \in E} \oplus^{iter \in 1 \sim t} \Delta_j^{(iter)}; \quad j_{1 \ldots k} \rightarrow i \in E$$

As they have the same initial value $x_i^{(0)}$and $x_i^{(1)}$ and receive the same delta messages, they can obtain the same value after the *apply* operation at data coherency stage.

Thirdly, assume the edges $j_1 \rightarrow i$ use parallel-edge mode, each replica of x_i has this edge. And thus $\Delta_{j1}^{(t)}$ is local messages for x_{i1}, x_{i2} and x_{i3}, and isn't sent to remote replicas.

Now, we can conclude that the eager data coherency for replicas of a vertex is equal to the lazy data coherency for replicas. □

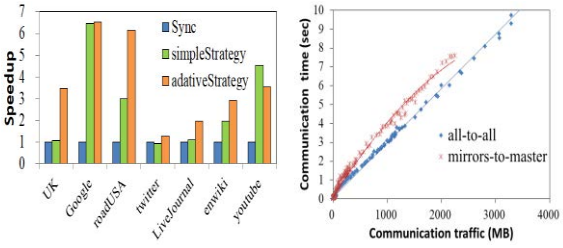
4 The LazyGraph System

In this section, we build a distributed graph processing system, called LazyGraph, to implement the *LazyAsync* execution approach. LazyGraph extends PowerGraph system to support *LazyAsync* engine and exploits the graph-aware optimizations to gain high performance. LazyGraph has implemented *LazyBlockAsync* engine based on the Sync engine, and will implement *LazyVertexAsync* engine based on the Async engine in the future.

4.1 Graph Loading and Partitioning

In LazyGraph, each machine starts from loading a separate subset of the graph. The graph partitioning in LazyGraph consists of a vertex-cut partitioning algorithm and an edge splitter. After loading, the vertex-cut partitioning places the graph structure and data across multiple machines by evenly assigning edges to machines and allowing vertices to span machines. The vertex-cut partitioning algorithm can be one of random-cut, coordinated-cut, grid-cut and hybrid-cut. Then the edge splitter selects some edges to be parallel-edges and dispatches these parallel edges to where they should be.

The edge splitter has three key elements. 1) Parallel-edges selecting criterion. An edge connecting two high-degree vertices or an edge with low-out-degree source and low-degree target will be split into parallel-edges. The former helps rapid convergence of the local computation, and the latter saves transmission cost. 2) The number of parallel-edges PE_{high} and PE_{low} comes from the solution of the equations, $[PE_{high}*(P - 1) + PE_{low}*(P/3)] / P = TEPS * t_{extra}$ and $PE_{low} = PE_{high} * 550$, where PE_{high} is the number of high-degree parallel-edges, PE_{low} is the number of low-degree parallel-edges, P is the number of machines, TEPS is a 'traversed edges per second' rate and represents a machine performance, t_{extra} is an extra execution time introduced by parallel-edges and set by a user. The edge splitter determines the proportion of parallel-edges according to t_{extra}. 3) Dispatching parallel-edges follows the rule, that in the final distributed graph parallel-edges *v->u* must appear on all the machines replicas of *u* stored for unidirectional algorithms, or on all the machines all replicas of *v* and *u* stored for bidirectional algorithms. The edge splitter dispatches each parallel-edges *v->u*, until all parallel-edges don't violate this rule.



(a)Interval strategy on SSSP (b) communication modes

Figure 8. Graph-aware Optimizations

4.2 Graph-Aware LazyBlockAsync Engine

LazyGraph implements the *LazyBlockAsync* execution model defined in Section 3.4 and exploits graph-aware optimizations to gain high performance.

4.2.1 Adaptive Interval Between Two Adjacent Data Coherency Points

How long the data coherency for replicas should be delayed? To answer this question, a challenge faced by LazyGraph is input and algorithm sensitivity, where the best interval strategy may vary with different input sets and different algorithms. We adopt a machine learning technique to build an input-behavior-interval model, which predicts an optimal interval for an arbitrary input and algorithm.

The input-behavior-interval model includes two components, "is it a good time to turn on lazy mode?" and "how long does the local computation stage execute?" The first component is a classification problem, classifying the running status whether or not to turn on the lazy mode (turnOnLazy() function in Line 16 of Algorithm 1). We select decision trees method to learn the classification, and select the following features to train the model:

1) Locality of an input graph. We use E/V ratio and the replication factor λ (which is the average number of replicas for a vertex) to express the locality of a graph.

Table 1 shows λ of real-world graphs using coordinated-cut on 48 partitions. λ, from small to large, is road graphs, web graphs and social graphs.

2) Characteristic of a graph algorithm. Many graph algorithms proceed iteratively, updating the graph data in rounds until a fixpoint is reached. The number of active vertices, $vcnt^{t}$, is different on each iteration. We use a changing trend of the number of active vertices to describe the algorithm characteristic, $\text{trend} = (\text{vcnt}^{t-1} - \text{vcnt}^{t})/\text{vcnt}^{t-1}$. We count the number of active vertices $vcnt^{t}$ at each data coherency stage, if the *trend* is negative, the graph algorithm is in the ascent part; otherwise, in the descent part.
3) We set the first iteration without the local computation stage. After achieving $x_i^{(1)}$ and $\Delta_i^{(1)}$ based on $x_i^{(0)}$ at the first data coherency stage, LazyGraph begins executing a sequence of the two stages.

The second component of the input-behavior-interval model is how long does the local computation stage execute? The execution time of this stage should not be a fixed length. We collect the execution time *T* of the first iteration at each local computation stage online, and set execution time of this stage no more than *x*T* (doLC() function in Line 6 of Algorithm 1).

After training, the first component of the input-behavior-interval model is that the lazy mode is turned on when the controlling condition (*E/V <= 10 || (trend >= 0.07)*) is satisfied. And the second component is set as *3T*. The input-behavior-interval model means that if the locality of a graph is poor, the ascent part should synchronize frequently between replicas, and the descent part should synchronize rarely between replicas; but if the locali ty is good, both ascent and descent parts should synchronize rarely between replicas.

In Fig.8(a) we compare the performance of the adaptive interval strategy against a simple strategy, where the lazy mode always turns on and each local computation stage executes to convergence. The adaptive interval strategy does help LazyGraph gain high performance.

4.2.2 Communication Modes Switch

LazyGraph can dynamically switch between all-to-all and mirrors-to-master communication modes when exchanging delta messages at data coherency stage. All-to-all mode is appropriate for a small amount of communication traffic, and mirrors-to-master mode is appropriate for a large amount of traffic. After experiments (shown in Fig.8(b)), we observe that the communication time and the amount of traffic have a linear relationship for all-to-all mode, $t_{a2a} = 0.0029comm_{a2a} + 0.0848$; and a polynomial relationship for mirrors-to-master mode $t_{m2m} = -6*10^{-7}*comm_{m2m}^{2} + 0.0045*comm_{m2m} + 0.103$.

At data coherency stage, we use the following equations to calculate the communication volume as in:

$$comm_{a2a} = \sum_{v \in V} r_v^{hasDeltaMsg} * (rNum_v - 1) * sizeof(DeltaMsg)$$

$$comm_{m2m} = \sum_{v \in V} (r_v^{hasDeltaMsg} + rNum_v - 2) * sizeof(DeltaMsg)$$

where $r_v^{hasDeltaMsg}$ is the number of *v*'s replicas having deltaMsg, $rNum_v$ is the number of *v*'s replicas. And then we use two equations to estimate the communication times of all-to-all and mirrors-to-master modes, and select the faster one.

5 Evaluation

In this section, we compared the performance of the LazyGraph against Sync and Async of PowerGraph on a 48-node EC2-like cluster.

5.1 Experimental Methodology

We compare LazyGraph with the lazyBlockAsync engine against PowerGraph with Sync and Async engines, and report the average results of three runs for each experiment. The compiler used is GCC 4.8.1. All experiments are performed on a 48-node EC2-like cluster. Each node has 8 Intel Xeon cores, 32 GB of memory, and connected via 1 GigE Ethernet. We use coordinated vertex-cut partitioning algorithm to evaluate LazyAsync, Sync and Async.

Table 1 summarizes the large graphs used in our experiments. These real-world graphs were taken from the Stanford Large Network Dataset Collection [31], the Laboratory for Web Algorithmic [46], and the DIMACS shortest paths challenge [32].

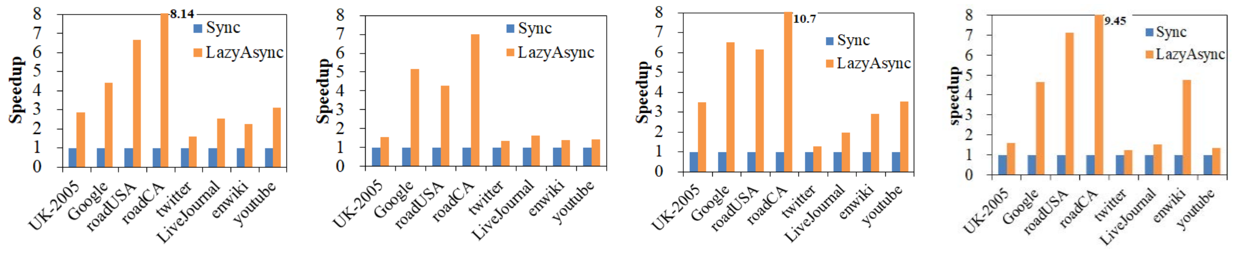
5.2 Performance

Fig.9 shows the overall speedup of LazyGraph and PowerGraph Sync for the four algorithms with different real-world graphs on 48 machines. LazyGraph outperforms PowerGraph Sync on all four algorithms: the speedups range from 1.25x to 10.69x across a variety of real-world graphs over PowerGraph, with an average speedup of 3.95x on k-Core, 3.1x on PageRank, 4.57x on SSSP and 3.91x on CC. A performance comparison between LazyGraph and PowerGraph Async is shown in Section 5.4.

The largest improvements are on the road graph, the smallest improvements are on the twitter graph, but for other graphs, the improvements are different on the four algorithms. For example, for UK-2005, compared with the PowerGraph Sync, LazyGraph achieves the speedups of 3.48x on SSSP and 1.49x on CC. But for com-youtube, compared with PowerGraph Sync, LazyGraph achieves the speedups of 3.54x on SSSP and 4.44x on CC.

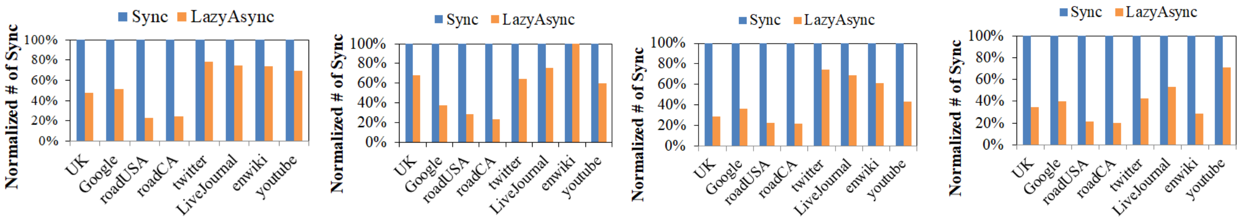
5.3 Explaining the Performance Improvement

The speedup of LazyGraph demonstrated in Fig.9 is due to reducing the number of global synchronizations and communication traffic. Since any changes to vertex data must be immediately communicated to all replicas of *v*, the eager data coherency approach leads to frequent global



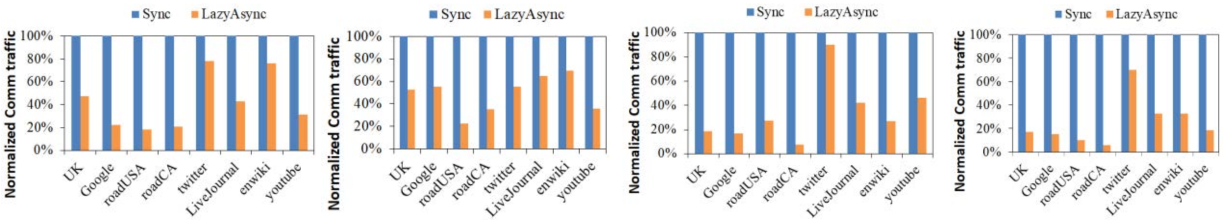
(a) k-core (b) PageRank (c) SSSP (d) CC

Figure 9. Speedup comparisons for k-Core, PageRank, SSSP and CC on real-world graphs on 48 machines



(a) k-core (b) PageRank (c) SSSP (d) CC

Figure 10. Normalized number of global synchronizations for k-Core, PageRank, SSSP and CC on 48 machines



(a) k-core (b) PageRank (c) SSSP (d) CC

Figure 11. Normalized Communication traffic for k-Core, PageRank, SSSP and CC on 48 machines

synchronization and communication. Fig.10 shows the number of global synchronizations for LazyGraph and PowerGraph Sync, normalized by PowerGraph Sync. Fig.11 shows the communication traffic for LazyGraph and PowerGraph Sync. The strong correlation between Fig.9 and Fig.10/Fig.11 illustrates that decreasing the number of global synchronizations and communication traffics leads to the performance improvement.

Note that in Fig.9, 10 and 11, the speedup rate of our approach largely depends on the replication factor λ of input graphs, and is independent of the graph sizes and the number of iterations. The lower λ of the input graph, the greater the speedup of LazyGraph. λ is 2.09 (roadNet-CA) < 2.14 (road-USA) < 2.47 (web-Google) < 2.7 (com-youtube) < 3.51 (UK-2005) < 4.96 (soc-LiveJournal) < 5.52 (twitter) < 7.22 (enwiki), respectively.

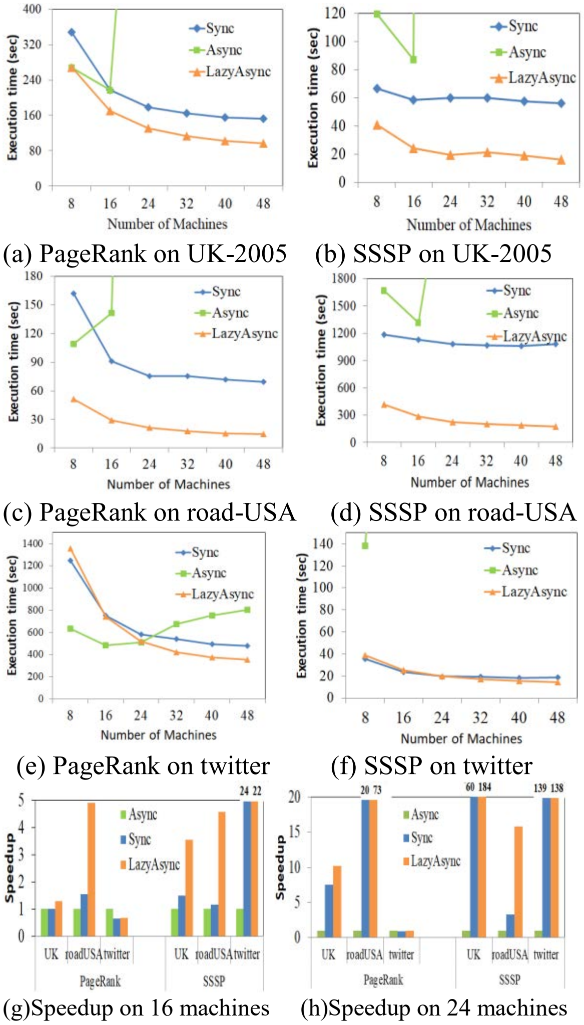
Table 1. Real-world graphs used for evaluation

	Graph	#V	#E	E/V	λ
web	UK-2005	40M	936M	23.73	3.51
	web-Google	0.9M	5.1M	5.83	2.47
road	road_USA_net	24M	58M	2.44	2.14
	roadNet-CA	2M	5.5M	2.82	2.09
social	twitter	61.58M	1468M	23.85	5.52
	soc-LiveJournal	4.84M	68.9M	14.23	4.96
	enwiki	4.2M	101.36M	24.09	7.22
	com-youtube	1.1M	6M	5.27	2.70

5.4 Scalability

We evaluate the scalability of LazyGraph, PowerGraph Sync and Async with the increasing number of machines on PageRank and SSSP algorithms with UK-2005, road-USA, and twitter graphs. These three graphs represent web, road and social networks, respectively. As shown in Fig.12(a-f), LazyGraph has a good scalability when the machines number increases. Note that in Fig.12(e), PowerGraph Async does scale with machines on PageRank with twitter graph, but gets performance degradation on SSSP and PageRank with web and road graphs, when the machine number is larger than 16. Fig.12(g)(h) show the speedups of LazyGraph, PowerGraph Sync and Async for PageRank and SSSP with these three graphs on 16 machines and 24 machines. LazyAsync has a better scalability than Async.

6 Related Work



(a) PageRank on UK-2005 (b) SSSP on UK-2005

(c) PageRank on road-USA (d) SSSP on road-USA

(e) PageRank on twitter (f) SSSP on twitter

(g)Speedup on 16 machines (h)Speedup on 24 machines

Figure 12. (a-f) Comparison between PowerGraph and LazyGraph for PageRank and SSSP on increasing machines. (g,h) Speedup comparison on 16 machines and

A large number of graph-parallel systems [1-19] have been proposed to process large-scale graphs. Pregel[1], Giraph[6], Cyclops[48], GraphX[5] and Gemini[4] adopt the Bulk Synchronous Parallel (BSP) model which has a global synchronization after each iteration. GraphLab[2], PowerGraph[3], PowerLyra[8], Trinity[7] and GRACE[19] provide Sync and Async engines to process a graph. PowerSwitch allows dynamic switching between sync and async engines to gain optimal performance. However, all of them use eager data coherency for replicas of a vertex, which leads to frequent global synchronization and communication. LazyGraph uses the lazy data coherency approach *LazyAsync* to solve this problem.

Recently, Hieroglyph[52] enables each replica to independently update its local data as well. But there are three differences between Hieroglyph and LazyGraph. 1) Hieroglyph focuses on decoupling computations from communications to obtain locally sufficient computation; LazyGraph focuses on delaying the data coherency between replicas to reduce the number of global synchronizations and network traffic. 2) In Hieroglyph, user-defined functions resolve the inconsistency. In LazyGraph, the system automatically maintains the data coherency for replicas. 3) LazyGraph supports two transmission modes. Hieroglyph supports one-edge mode.

Many works have attempted to optimize distributed graph processing systems from graph partitioning [3,4,8,33,20-25] to reduce communication cost and load imbalance, overlapping communication and computation [4,34,35] to achieving scalability, and accelerating computation by using multi-core[15-19,55,56] and GPU[44,45,54]. If putting these optimizations and the *LazyAsync* execution model together, the distributed graph system will achieve good efficiency and scalability.

There are other works that focus on graph querying systems [36-38], machine learning and data mining systems [43], temporal analytics [39-42], and streaming processing systems [47]. For distributed parallel graph algorithms, it could also be beneficial to apply lazy data coherency approach *LazyAsync* in LazyGraph to boost performance.

7 Conclusion

In this paper, we propose a lazy data coherency approach, called *LazyAsync*, which treats replicas of a vertex as independent vertices and maintains the data coherency by computations, rather than communications in existing eager approach. *LazyAsync* delays the data coherency between replicas of a vertex, asynchronously executes replicas of a vertex and reduces the number of global synchronization and network traffic. Based on PowerGraph, we develop a distributed graph processing system called LazyGraph that 1) uses the *LazyAsync* approach as the execution model to reduce the number of global synchronization and communication; 2) supports two message transmission modes at the same time to benefit from *one-edge* mode on saving computation and *parallel-edges* mode on saving transmission cost; 3) exploits graph-aware optimizations including adaptive interval strategy between two adjacent data coherency points, dynamic switching between all-to-all and mirrors-to-master communication modes. On a 48-node EC2-like cluster, LazyGraph outperforms PowerGraph across a variety of real-world graphs, with a speedup ranging from 1.25x to 10.69x.

Acknowledgments

We thank all the reviewers for their valuable comments and suggestions. This work was supported in part by the National Key Research and Development Program of China (2017YFB0202002), the National Natural Science Foundation of China (61402445, 61521092, 61432016, 61432018, 61332009, U1736208).

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD 2010), ACM, pp. 135-146, 2010.

[2] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed Graphlab: a Framework for Machine Learning and Data Mining in the Cloud. In Proceedings of the VLDB Endowment, pp. 716-727, 2012.

[3] Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI, 2012), pp. 17-30, 2012.

[4] X. Zhu, W. Chen, W. Zheng and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In Proceedings of t 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 301-316, 2016.

[5] 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 Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI 2014), pp. 599-613, 2014.

[6] C. AVERY. Giraph: Large-scale graph processing infrastructure on hadoop. In Proceedings of the Hadoop Summit, 2011.

[7] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on A Memory Cloud. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013), ACM, pp. 505-516, 2013.

[8] R. Chen, J. Shi, Y. Chen, and H. Chen。 Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys 2015), 2015.

[9] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or Async: Time to Fuse for Distributed Graph-Parallel Computation. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015), 50(8), pp. 194-204, 2015.

[10] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015), ACM, pp. 410-424, 2015.

[11] S. Seo, E. J. Yoon, J. Kim, and S. Jin。 Hama: An Efficient Matrix Computation with the Mapreduce Framework. In Proceedings of 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom 2010), IEEE, pp. 721-726, 2010.

[12] D. Gregor, and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In Parallel Object-Oriented Scientific Computing, 2015.

[13] I. Hoque, and I. Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (SIGOPS 2013), 2013.

[14] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A System for Distributed Graph Mining. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015). pp. 425-440, 2015.

[15] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In Proceedings of the TwentyFourth ACM Symposium on Operating Systems Principles (SOSP 2013), pp. 456-471, 2013.

[16] J. Shun, and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2013), 48(8), pp. 135-146, 2013.

[17] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. In Proceedings of the VLDB Endowment (VLDB 2015), 8(11), pp. 1214-1225, 2015.

[18] K. Zhang, R. Chen, and H. Chen. NUMA-Aware Graph-Structured Analytics. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015), pp. 183-193, 2015.

[19] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In CIDR. 2013.

[20] U. V. Catalyurek, and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix Vector Multiplication. In Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR 1996), pp. 75-86, 1996.

[21] N. Jain, G. Liao, and T. L. Willke. GraphBuilder: A Scalable Graph ETL Framework. In First International Workshop on Graph Data Management Experiences and Systems (GRADES 2013), 2013.

[22] G. Karypis and V. Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. In Proceedings of the 1996 ACM/IEEE conference on Supercomputing, 41(2), pp. 278-300, 1999.

[23] K. Schloegel, G. Karypis, and V. Kumar. Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning. In Proceedings of the 1998 ACM/IEEE conference on Supercomputing (Euro-Par 2000), pp. 296-310, 2000.

[24] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2012), pp 1222-1230, 2012.

[25] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM 2014), pp.333-342, 2014.

[26] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In Proceedings of 19th International World-Wide Web Conference (WWW 2010), pp. 591-600, 2010.

[27] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. In Journal of Software: Practice and Experience, 34(8), pp. 711-726, 2004.

[28] H. Haselgrove. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm, 2010.

[29] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On Compressing Social Networks. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2009), pp. 219-228, 2009.

[30] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics, 6(1), pp. 29-123, 2009.

[31] SNAP: Stanford Network Analysis Platform. snap.stanford.edu/snap/index.html

[32] 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml.

[33] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced Graph Edge Partition. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (SIGKDD 2014), pp. 1456-1465, 2014.

[34] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling Graph Computation to the Trillions. In Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC 2015), pp. 408-421, 2015.

[35] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. Pgx.d: A Fast Distributed Graph Processing Engine. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15), 2015.

[36] A. Quamar, A. Deshpande, and J. Lin. Nscale: Neighborhood-Centric Analytics on Large graphs. In Proceedings of the VLDB Endowment, pp.1673-1676, 2014.

[37] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016.

[38] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST15), pp. 45-58, 2015.

[39] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 2012), pp. 85-98, 2012.

[40] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In Proceedings of the Ninth European Conference on Computer Systems(EuroSys 2014), 2014.

[41] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In Proceedings of 2013 IEEE 29th International Conference on Data Engineering (ICDE 2013), pp. 997-1008, 2013.

[42] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In Proceedings of IEEE 31st International Conference on Data Engineering (ICDE 2015), 2015.

[43] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the Hidden Dimension in Graph Processing. In Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI16), 2016.

[44] J. Zhong, and B. He. Medusa: Simplified Graph Processing on GPUs. In IEEE Transactions on Parallel and Distributed Systems (TPDS 2013). 25(6), pp.1543-1552, 2013.

[45] J. Zhong, and B. He. Parallel Graph Processing on Graphics Processors Made Easy. In Proceedings of the VLDB Endowment (VLDB 2013), 2013.

[46] The laboratory for web algorithmic. http://law.dsi.unimi.it/datasets.php.

[47] D. G. Murray, F. Mcsherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 2013), 2013.

[48] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and Communication Efficient Graph Processing with Distributed Immutable View. In Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC 2014), 2014.

[49] S. Brin, and L. Page. The Anatomy of A Large-Scale Hypertextual Web Search Engine. In Proceedings of Seventh International World-Wide Web Conference (WWW 1998), 1998.

[50] Gonzalez J. E., Low Y., Guestrin C., and O'HALLARON, D. Distributed Parallel Inference on Large Factor Graphs. In Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI 2009). pp. 203-212, 2009.

[51] M. Han, and K. Daudjee Giraph. Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. In Proceedings of the VLDB Endowment. 2015.

[52] X. Ju, H. Jamjoom, K. G. Shin. Hieroglyph: Locally-Sufficient Graph Processing via Compute-Sync-Merg. In Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, 2017.

[53] R. R. McCune, T. Weninger, and G. Madey. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. In ACM Computing Surveys, 48(2), 2015.

[54] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, H. Jin. Frog: Asynchronous Graph Processing on GPU with Hybrid Coloring Model. In IEEE Transactions on Knowledge and Data Engineering, 30 (1), pp 29-42, 2018.

[55] L. Wang, F. Yang, L. Zhuang, H. Cui, F. Lv, X. Feng. Articulation Points Guided Redundancy Elimination for Betweenness Centrality. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2016), 51(8), 2016.

[56] J. Zhao, H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang. An Empirical Model for Predicting Cross-core Performance Interference on Multicore Processors. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT 2013), pp. 201-212, 2013.