Front. Comput. Sci.
https://doi.org/10.1007/s11704-018-6400-1

RESEARCH ARTICLE

A communication-reduced and computation-balanced framework for fast graph computation

Yongli Cheng[1], Fang Wang (✉)[2,3], Hong Jiang[4], Yu Hua[2,3], Dan Feng[2,3], Lingling Zhang[2,3], Jun Zhou[2,3]

1 College of Mathematics and Computer Science, FuZhou University, Fuzhou 350116, China
2 Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
3 Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518300, China
4 Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, TX 76019, USA

Abstract The bulk synchronous parallel (BSP) model is very user friendly for coding and debugging parallel graph algorithms. However, existing BSP-based distributed graph-processing frameworks, such as Pregel, GPS and Giraph, routinely suffer from high communication costs. These high communication costs mainly stem from the fine-grained message-passing communication model. In order to address this problem, we propose a new computation model with low communication costs, called LCC-BSP. We use this model to design and implement a high-performance distributed graph-processing framework called LCC-Graph. This framework eliminates high communication costs in existing distributed graph-processing frameworks. Moreover, LCC-Graph also balances the computation workloads among all compute nodes by optimizing graph partitioning, significantly reducing the computation time for each superstep. Evaluation of LCC-Graph on a 32-node cluster, driven by real-world graph datasets, shows that it significantly outperforms existing distributed graph-processing frameworks in terms of runtime, particularly when the system is supported by a high-bandwidth network. For example, LCC-Graph achieves an order of magnitude performance improvement over GPS and GraphLab.

Received August 4, 2016; accepted September 7, 2017

E-mail: wangfang@hust.edu.cn

Keywords graph computation, communication decrease, computation balance

1 Introduction

Due to the increasing need to analyze, process and mine large real-world graphs, such as social networks, web graphs, chemical compounds and biological structures, there has been a recent surge of interest in distributed graph-processing frameworks in both academia and industry. Several Bulk Synchronous Parallel (BSP) computation model [1] based distributed graph-processing frameworks, such as Pregel [2], GPS [3], Giraph [4] and Grace [5], have been proposed to handle large-scale graphs.

The BSP-based distributed graph-processing frameworks, such as Pregel, GPS and Giraph, typically employ a think-like-a-vertex programming model to support iterative graph computation, which considers a graph-computing job as a set of iterations called supersteps. In each superstep, a user-defined **Compute(v)** function is invoked for each vertex **v**, conceptually in parallel. In fact, due to the limited core count of each compute node, the executions of vertices are assigned to a limited number of work threads that execute concurrently. All work threads of compute nodes start simultaneously. Each work thread loops through its assigned vertices by using the **Compute(v)** function. Each vertex updates its

state based on incoming messages from its neighboring vertices and then sends messages, generated based on the new state and the incoming messages, to its neighbors. The generated messages are available only in the next superstep, and there is a barrier at the end of each superstep to ensure that all generated messages are delivered to their destination vertices before the next superstep starts. The graph-computing job stops when all vertices vote to stop the computation. This computation model has several significant advantages that can be described as follows.

Determinism: The barriers between any two consecutive supersteps guarantee that a graph-computing job always outputs the same result from different executions.

Ease of Programming: Graph algorithms can be encoded using the think-like-a-vertex philosophy. Graph algorithm programmers only need to focus on the logic of a given graph algorithm within the individual vertex computation functions, **Compute(v)**.

High Scalability: In each superstep, vertices of an input graph are executed independently. Hence, vertices can be assigned to more compute nodes to improve the run time performance of a graph-computing job.

However, BSP-based distributed graph-processing frameworks routinely suffer from high communication costs [4, 6–10]. The communication cost is defined as the time for vertices to interact with each other, including the communication time for sending messages through the network and the extra sender-side and receiver-side communication overheads. Within each superstep, vertices interact with each other by passing messages. This fine-grained message-passing communication model is inefficient and severely underutilizes the network bandwidth capacity even when the *message buffer* technique [2, 3] is used to amortize the average overhead of each message. For example, our experiments show that when GPS [3] performs the PageRank [11] algorithm with the *message buffer* technique on a 40Gbps InfiniBand network, the communication cost is responsible for 95% of the overall run time while only 0.9% of the network bandwidth is utilized. The high communication costs stem mainly from following four factors, i.e., the bulk of the extra communication volume, data copying overhead both at sender side and receiver side, the parsing overhead at receiver side and the poor communication bandwidth utilization, as detailed in Section 2.

In this paper, we propose a high-performance distributed in-memory graph-processing framework, called LCC-Graph. The high performance of LCC-Graph stems from the low communication costs and reduced computation time. By designing and implementing LCC-Graph, this paper makes the following five contributions.

1) *A computation model with low communication costs*, called LCC-BSP, that decomposes each superstep into two distinct steps of computation and communication. In the computation step, each vertex does computation task by reading and writing its edge values directly, reducing the computation time. In the communication step, each compute node exchanges full remote out-edge data blocks with other compute nodes to implement edge-value transfers among inter-node vertices. The clear advantage of this decomposition of superstep is that the interactions among vertices will be finished instantaneously and simultaneously in a well-orchestrated concurrent manner after the computation step that runs for only a short period of time.
2) *An edge-block based data representation.* By organizing the edge values of each compute node intelligently in the preprocessing phase, this data representation provides four salient advantages that enable the reduced communication costs and computation time in LCC-BSP. First, it eliminates the high extra volume of communication in existing BSP-based distributed graph-processing frameworks. Second, it avoids the data copying and message batch parsing overheads. Third, it enables the network bandwidth capacity to be efficiently utilized. Finally, the computation workload of each vertex is reduced.
3) *Two optimizations.* First, we optimize the network ecosystem to further utilize the network bandwidth capacity more efficiently. In the context of this paper, the network ecosystem is interpreted as the combination of the network hardware and communication protocol software. Second, an out-edge data block compression scheme is introduced to improve the performance of those graph algorithms with a relatively small number of interactions among vertices, despite of the fact that most graph algorithm jobs have a very large number of interactions among vertices [11–13].
4) *The Improved, Compute-Aware SGP (CA-SGP) graph partitioning method.* The computation imbalance problem becomes the new bottleneck for LCC-Graph since the communication time has been significantly reduced by our LCC-BSP computation model. By using this method, LCC-Graph balances the computation workload among compute nodes, gaining notable reductions in computation time for each superstep.
5) *The design and prototype implementation of LCC-*

Graph. LCC-Graph also flexibly supports some peculiar requirements of graph algorithms even if these cases rarely happen. These requirements include messages sent to non-neighbor vertices and multiple messages sent to a single destination vertex.

The rest of the paper is structured as follows. Background and motivation are presented in Section 2. Section 3 introduces our LCC-BSP, a high-performance computation model. Section 4 presents LCC-Graph, our LCC-BSP based graph-processing framework. The Improved, Compute-Aware SGP (CA-SGP) graph partitioning method is presented in Section 5. Experimental evaluations of the LCC-Graph prototype are presented in Section 6. We discuss related work in Section 7 and conclude this paper in Section 8.

2 Background and motivation

In this section, we first discuss the efficiency issues of the BSP-based distributed graph-processing frameworks. We then introduce GraphChi, a single-node disk-based graph-processing framework. GraphChi has four salient features that cause the reduction of the disk I/O costs. The insights gained through the efficiency issues of existing BSP-based distributed graph-processing frameworks and the features of GraphChi help motivate us to propose an LCC-BSP computation model that eliminates the high communication costs incurred by existing BSP-based distributed graph-processing frameworks, and reduces the computation workload of each vertex.

2.1 BSP-based graph-processing frameworks

As mentioned before, although BSP-based distributed graph-processing frameworks, such as Pregel [2] and GPS [3], leverage the *message buffering* technique to amortize the average overhead of each message, they continue to suffer from the high communication costs [4, 6–10]. Figure 1 shows the execution process of any pair of compute nodes in the BSP-based distributed graph-processing frameworks with the *message buffering* technique. The main reasons for the high communication costs are fourfold.

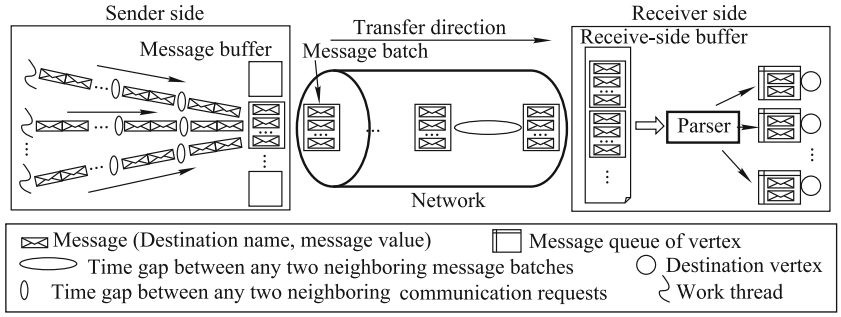


Fig. 1 The execution process of any pair of compute nodes in BSP-based distributed graph-processing frameworks with message buffering technique

The bulk of the extra communication volume comes from the need to carry the destination vertex name on each message. By using the name of the destination vertex on each message, the receiver side can identify which destination vertex the message belongs to, and then dispatches the message to the message queue of the destination vertex [2, 3]. For most graph algorithms, such as Label Propagation [14], Single-Source Shortest-Paths [6] and PageRank [11], a message usually carries a value with a short size, such as a 4-byte integer or floating-point number used to store the label value, shortest-path value or pagerank value. However, the size of the name of each destination vertex is usually longer than that of the message value (i.e., payload). For example, in order to handle the graphs with more than four billion vertices, graph-processing frameworks must use an 8-byte long-integer number to identify each vertex. In this case, the extra communication volume due to vertex names is responsible for 67% of the overall communication volume.

Data copying overhead. At the sender side, any message generated by a work thread is first sent to the message buffers [2, 3]. When a message buffer is filled up, the message batch in the message buffer is sent to the network. At the receiver side, when a message batch is received by the *message parser*, it parses the message batch and enqueues the messages in the message batch to the message queues of the destination vertices according to the name of each destination vertex [2,3]. Thus, each vertex can identify the messages sent to itself. There are two rounds of data copying, one at the sender side and the other at the receiver side.

The parsing overhead. At the receiver side, in order to dis-

patch the messages in the message batches to the message queues of the destinations vertices, the *message parser* parses the message batches, resulting in the high parsing overhead [3].

The poor communication bandwidth utilization stems from the slow process of message generation and the general-purpose communication protocols that only provide a limited bandwidth. First, the message buffers are filled up rather slowly because any two consecutive buffer-filling operations of a work thread are separated by the three distinctive tasks of computation, message generation and decision on the placement of each message. Therefore, it will take a relatively long time to fill up a message buffer of a reasonably large size to achieve a good amortization, which causes a long idle period of the network. In fact, the larger the size of the message buffers is, the longer the idle periods of the network are. Hence, the size of the message buffer is a tradeoff between the gains from the batched communication and the idle periods of the network. Second, existing distributed graph-processing frameworks are usually designed on the basis of the general-purpose communication protocols that only provide a limited bandwidth. This limit hinders these frameworks from speeding up the transfers of the batched messages, particularly when the system is supported by a high-bandwidth network.

2.2 GraphChi

GraphChi [7] is a single-node disk-based graph processing framework. It introduces a novel mechanism called Parallel Sliding Windows (PSW) to reduce the disk I/O costs to improve performance. GraphChi works as follows. In preprocessing stage, the vertices of the input graph are divided into continuous but disjoint $\boldsymbol{P}$ intervals, each of which is associated with a shard. Each **shard** is a disk file that stores all the edges along with their values that have their destination vertex labels in a given interval. Edges are stored in order of their *sources*. This data representation has a clear advantage, that is, for any loaded subgraph $\boldsymbol{S}$, shard $\boldsymbol{S}$ contains the information of all the in-edges and local out-edges of the loaded subgraph, and all the out-edges with their destination vertices in a given unloaded subgraph $\boldsymbol{R}$ can be obtained from a contiguous disk file chunk in the shard $\boldsymbol{R}$. In each iteration, GraphChi executes $\boldsymbol{P}$ subgraphs sequentially. Each execution process of a subgraph $\boldsymbol{S}$ consists of three stages. (1) loading subgraph from disk. GraphChi obtains all the in-edges and local out-edges by reading the full shard $\boldsymbol{S}$ and other out-edges by reading $\boldsymbol{P}$-1 contiguous disk file chunks in other $\boldsymbol{P}$-1 shards. (2) executing the user-defined **Update(v)** function for each vertex **v** in the loaded subgraph. (3) saving results to disk.

There are four salient features of PSW. First, it minimizes the disk I/O workloads. In the stage of saving results, GraphChi only needs to write the edge values back to the disk files (shards) since only the edge values are updated during computation [7]. However, in existing distributed graph-processing frameworks, each message consists of a destination name and a message value. Second, the edge values in the edge-data blocks indexed by the vertices are read and written by the **Update(v)** function directly. Thus, the saving results stage can be executed immediately after the computation stage is finished. Third, by reading/writing the full shard and the contiguous disk file chunks, it alleviates random accesses to improve the I/O performance. Finally, the computation time of each vertex is reduced to the time for reading/writing its in-edge/out-edge values only. However, in existing distributed graph-processing frameworks, the computation of each vertex requires the extra time for identifying which message buffer each message belongs to and then sending the message to the message buffer [3].

However, the PSW of GraphChi is not suitable for and cannot be used in distributed graph-processing frameworks. Because the shards are tightly coupled by PSW, that is, in order to load a subgraph $\boldsymbol{S}$ from the disk, GraphChi needs to obtain not only the in-edges and local out-edges of this subgraph from shard $\boldsymbol{S}$, but also other out-edges of this subgraph from $\boldsymbol{P}$-1 contiguous disk file chunks in other $\boldsymbol{P}$-1 shards.

2.3 Motivation

The efficiency issues of existing BSP-based distributed graph-processing frameworks and the salient features of GraphChi help motivate us to propose a high-performance computation model with low communication costs and reduced computation time, called LCC-BSP. The high performance of LCC-BSP stems from four key features. (1) It eliminates the high extra volume of communication. (2) It avoids the data copying and the message batch parsing overheads. (3) It enables the network bandwidth capacity to be utilized efficiently. (4) The computation work of each vertex is reduced.

3 LCC-BSP computation model

As mentioned before, the BSP-based distributed graph- processing frameworks typically employ a think-like-a-vertex programming model to support iterative graph computation,

which considers a graph-computing job as a set of iterations called supersteps. LCC-BSP, like the computation model of BSP-based distributed graph- processing frameworks, considers a graph-computing job as a set of supersteps. However, unlike BSP-based distributed graph- processing frameworks, the communication process in LCC-BSP is decoupled from its computation process by explicitly dividing each superstep into a computation step and a communication step.

In the computation step, a user-defined **Update(v)** function is invoked for each vertex **v** in parallel. Inside **Update(v)**, the vertex **v** updates its state by its in-edge values and then updates its out-edge values. The in-edge values of vertex **v** were updated by the source vertices of the in-edges in the previous superstep, and the out-edge values of vertex **v** will be used by the destination vertices of the out-edges in the next superstep.

In the communication step, edge values are moved to implement the interactions among vertices, since the out-edges of a vertex are the in-edges of its neighboring vertices. For each compute node, all out-edge values for a given remote compute node are organized into a single full remote out-edge data block in the preprocessing phase judiciously. Similarly, all in-edge values for a given remote compute node are organized into a single full remote in-edge data block. A remote out-edge data block is an exact copy of a remote in-edge data block of a remote compute node. After the computation step has finished, each compute node only needs to send P-1 remote out-edge data blocks to P-1 remote compute nodes simultaneously, to update their respective copies, where P is the number of compute nodes.

In order to ensure the determinism of graph algorithms, there are two measures. First, in each compute node, a local synchronous barrier is added between the computation step and the communication step. It ensures that all out-edge values are updated completely before sending the out-edge data blocks to remote compute nodes. Second, at the end of each superstep, instead of using a global barrier, a local barrier in each compute node is required to ensure the determinism of graph-computing jobs. Thus, each compute node can start the next superstep before the expected out-edge data blocks have been received. Because each compute node has a certain number(P-1) of the expected out-edge data blocks.

3.1 Performance analysis

The clear advantage of this computation model is that, in each superstep, the interactions among vertices will be finished instantaneously and simultaneously in a well-orchestrated concurrent manner after the computation step that runs only a short period of time. Furthermore, the key advantage of the communication step is that each out-edge data block is an exact copy of an in-edge data block of a remote compute node, that is, each edge value has an identical and fixed position in the two copies. Using the fixed position, each edge value indexed by its vertex can be identified by the vertex directly in both the sender side and the receiver side without the vertex name, avoiding the high data copying and parsing overheads. As shown in Fig. 2, there are four salient features that contribute to the high performance of this computation model.

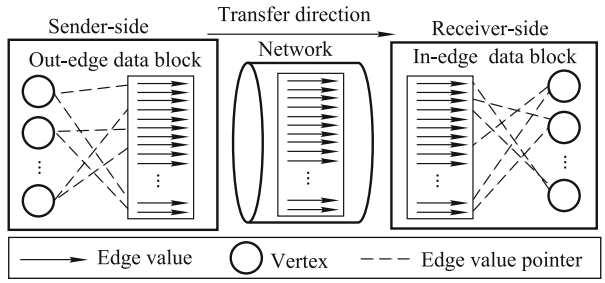


Fig. 2 The execution process of any pair of compute nodes in the LCC-BSP computation model

The reduced communication volume. LCC-BSP eliminates high extra volume of communication required to carry the name of destination vertex on each message in existing BSP-based distributed graph-processing frameworks. Since the out-edge data blocks that are sent to the network include the edge values only.

The eliminated data copying and parsing overheads. As mentioned before, the **Update(v)** function reads/writes the edge values directly in both sender side and the receiver side, avoiding the high data copying and parsing overheads.

The reduced computation time. In each computation step, the computation time is much shorter than that of the existing BSP-based distributed graph-processing frameworks, since the computation time of each vertex is reduced to the time for reading/writing its in-edge/out-edge values only, eliminating the time for message generation, decision on the placement of each message in the latter.

Highly efficient communication. First, in each superstep, LCC-Graph only causes a very short idle period of the network occupied by the computation step due to the reduced computation time. Second, in the communication step, each compute node only needs to send P-1 remote out-edge data blocks to P-1 remote compute nodes simultaneously. This well-orchestrated concurrent manner provides a sufficiently large instantaneous network workload that enables the network bandwidth capacity to be efficiently utilized, particularly when the system is supported by a high-bandwidth

network that is more easily accessible than ever before. Intuitively, existing BSP-based distributed graph-processing frameworks with message buffers can also provide a large instantaneous network workload by using large enough message buffers. However, larger message buffers cause longer idle periods of the network used by the work threads to fill up the message buffers due to the slow process of message generation, as discussed in Section 2. Moreover, the optimization of the network ecosystem, as presented in Section 4, helps further speed up the transfer of the out-edge data blocks.

Summary: There are two differences between LCC-BSP computation model and the computation model of BSP-based distributed graph-processing frameworks. First, in each superstep of the BSP-based distributed graph-processing frameworks, once the work threads of each compute node begin to execute their vertices by using the user-defined **Compute(*vertex v*)** function, the generated messages will be sent to the network during the supersetp. Unlike this computation model, the communication process in LCC-BSP is decoupled from its computation process by explicitly dividing each superstep into a computation step and a communication step. This decomposition enables the high performance of LCC-Graph, as mentioned before. Second, unlike BSP-based distributed graph-processing frameworks, at the end of each superstep, instead of using a global barrier, a local barrier in each compute node is required to ensure the determinism of graph-computing jobs. When the P-1 out-edge data blocks of each compute node have been received, the compute node can start the next superstep immediately, avoiding the overhead of the global barrier.

3.2 An example

To better illustrate the LCC-BSP computation model we use an example of the computation of PageRank [3]. In this example, the directed graph is organized into three subgraphs residing in three compute nodes. Each subgraph includes a local edge data block (**LDB**), two remote in-edge data blocks (**RIDBs**) and two remote out-edge data blocks (**RODBs**). Figure 3 shows a superstep of LCC-BSP computation model processing the PageRank algorithm.

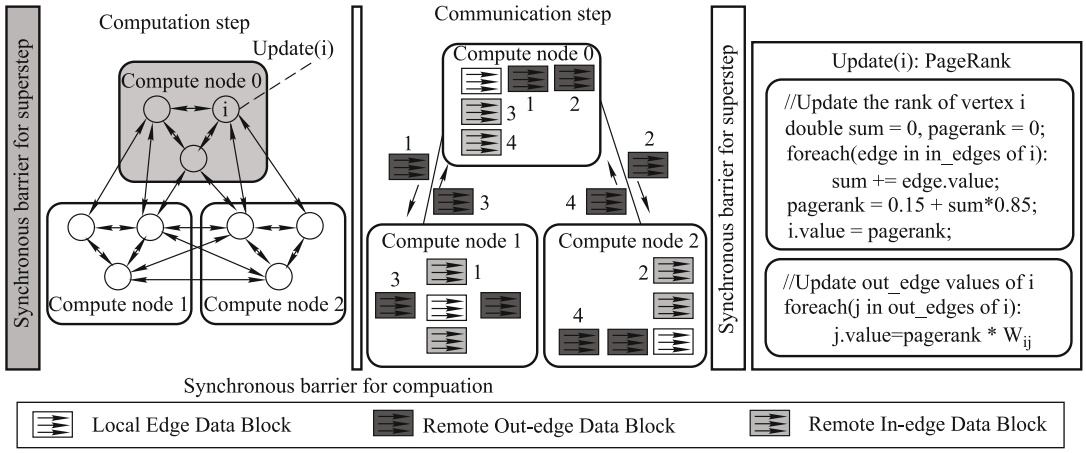


Fig. 3 An example of LCC-BSP computation model

In the computation step, each vertex **v** is executed by using the **Update(v):PageRank** function concurrently with other vertices. Consider vertex **i** as an example. First, a new *pagerank* is calculated according to the in-edge values of vertex **i**. Then the value of vertex **i** is updated with the new *pagerank*. Finally the out-edges' values of vertex **i** are updated based on new *pagerank*. The edge values, organized into edge data blocks, are read and written by the **Update(v): PageRank** function directly, avoiding data copyings. In the communication step, two remote out-edge data blocks of each compute node are sent to two other compute nodes, to update their respective copies. In this example, compute node 0 sends the block 1 to compute node 1 and the block 2 to compute node 2. Meanwhile each compute node will receive two remote out-edge data blocks from two other compute nodes to update its corresponding remote in-edge data blocks. In this example, compute node 0 receives the block 3 from compute node 1 and the block 4 from compute node 2.

3.3 Challenges

There are two key challenges in the implementation of the LCC-BSP computation model. The first one is to organize all the out-edge/in-edge values from a given compute node to

another into a single remote out-edge/in-edge data block. The other is to guarantee that each out-edge data block is an exact copy of an in-edge data block of a remote compute node. To address these challenges, the new *edge-block based data representation* is introduced, as presented in Section 4.1.

4 LCC-Graph framework

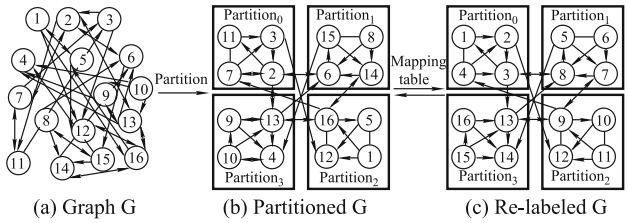
In this section, we present LCC-Graph. Key components and unique features of the LCC-Graph are detailed in various subsections next.

4.1 The edge-block based data representation

In this subsection, we present the edge-block based data representation that provides four salient features of the LCC-BSP computation model. We will now describe how the input graph is partitioned and organized into edge-block based subgraphs during the preprocessing phase. Each subgraph resides in the memory of a compute node.

Definitions: Let G=(V,E) denote an input graph with its vertex set V and edge set E, and let $\text{Partition}_0 \cup \text{Partition}_1 \cup \cdots \cup \text{Partition}_{P\text{-}1} = V$ be the P partitions of V, where $\text{Partition}_i \cap \text{Partition}_j = \emptyset$ $(i \neq j)$. For each Partition_i, the vertices in Partition_i, along with their edges, are defined as a subgraph of the input graph. The input is a directed graph and an undirected graph can be treated as a directed one by considering each undirected edge as two opposite directed edges.

Graph partitioning: By using a user-specified graph partitioning method, the input graph is divided into P partitions, as depicted in Fig. 4(b), where P is the number of compute nodes.



(a) Graph G (b) Partitioned G (c) Re-labeled G

Fig. 4 An example of graph partitioning and re-labeling vertices. Circles denote vertices with their labels. Note that each partition is a subset of vertices in graph G (a). To help us better understand the subgraph construction process, the edges are also illustrated in (b) and (c)

Relabeling vertices: It is assumed that the vertices are labeled from 1 to |V|. After dividing the input graph into P partitions, the vertices of the graph are unordered again. We relabel vertices of the P partitions sequentially to make the labels of vertices form P continuous but disjoint intervals, corresponding to the P partitions. Consider the re-labeled graph G as an example, shown in Fig. 4(c), the labels of vertices consist of four continuous but disjoint intervals, i.e., interval_0: [1–4], interval_1: [5–8], interval_2: [9–12] and interval_3: [13–16], corresponding to Partition_0, Partition_1, Partition_2 and Partition_3 respectively. By using the continuous but disjoint intervals, it is easy to identify which partition the vertices belong to, instead of accessing the global graph-partitioning information. This measure further enables the judicious data representation, described next. The relabeling has a negligible overhead since this process done in parallel on each compute node.

Shards: A **shard** is a consecutive memory chunk that stores all the edges along with their values that have their destination vertex labels in a given interval. That is, a shard is associated with each interval. Then, the edges in each shard are sorted by the *sources*. Shards are labeled from 0 to P-1. As shown in Fig. 5, four shards are generated for the four intervals, labeled as shard_0, shard_1, shard_2 and shard_3 respectively. Due to the sorting overhead, the sharding time is responsible for ~27.61% of the overall preprocessing time, as shown in Section 6.

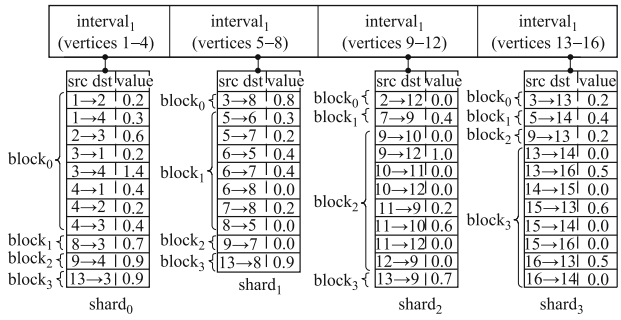


Fig. 5 Shards of re-labeled graph G (Each row of edge blocks denotes a directed edge with its value)

Note that the "8↔3" notation shown in Fig. 4(c) indicates that there are two directed edges, "8→3" and "3→8". Since the destination vertex "3" and "8" reside within interval_0 of [1-4] and interval_1 of [5-8] respectively, the directed edges "8→3" and "3→8" with their values are stored in shard_0 and shard_1 respectively. The other bidirectional edges, e.g., "13↔9", have similar meanings.

Constructing subgraphs: In order to create a subgraph_p for the vertices in Partition_p, where $0 \leqslant p \leqslant P\text{-}1$ and P is the number of compute nodes, in-edges and out-edges of these vertices need to be obtained. First, shard_p contains the information of all the in-edges and local out-edges (their *destinations* are also in Partition_p) for the vertices in Partition_p, so the in-edges and local out-edges can be eas-

ily obtained from shard$_p$. Consider the re-labeled Graph G, shown in Fig. 4(c), vertex "2" residing in Partition$_0$ has two in-edges ("1→2" and "4→2") and a local out-edge ("2→3"), which are stored in the first row, seventh row and the third row in the shard$_0$ respectively, as shown in Figure 5. Second, since the out-edges of a given vertex are the in-edges of its neighboring vertices, the remote out-edges of the vertex are stored in the other P-1 shards. Consider the vertex "2", it has a remote out-edge("2→12") which is stored in the first row in shard$_2$. Moreover, since all the labels of the vertices in Partition$_p$ are within the interval$_p$ and the edges in each shard are stored in order of their *sources*, the remote out-edges of the vertices in Partition$_p$ with their *destinations* within a given remote Partition$_j$ are stored in a contiguous memory chunk in the shard$_j$. Hence, the full remote out-edges of the vertices in Partition$_p$ can be obtained from the P-1 contiguous memory chunks in the other P-1 shards. However, in a distributed setting, it is inefficient to deploy P shards on all compute nodes of the cluster. Thus, each shard is decomposed into P edge blocks (labeled 0 to P-1) according to the interval that contains the source vertex labels corresponding to the edges. As shown in Fig. 5, each shard is split into four edge blocks: block$_0$, block$_1$, block$_2$ and block$_3$. There are P^2 edge blocks in total, and we use B(x, y) to identify an edge block, with block label x and shard label y. Consider Fig. 5 as an example, in order to construct subgraph$_0$, all the in-edges and local out-edges of vertices in Partition$_0$ can be obtained first by a full reading of the shard$_0$ (from block$_0$ to block$_3$).Then all the remote out-edges can be obtained from the block$_0$ of shard$_1$, block$_0$ of shard$_2$ and block$_0$ of shard$_3$ respectively.

After blocking, we can construct P subgraphs for P compute nodes. Each subgraph$_p$ consists of a local-edge block (**LB**) B(block$_p$, shard$_p$), P-1 remote in-edge blocks (**RIBs**) B(block$_i$, shard$_p$) and P-1 remote out-edge blocks (**ROBs**) B(block$_p$, shard$_j$), where $0 \leqslant i \leqslant P$-1, $0 \leqslant j \leqslant P$-1, $i \neq p, j \neq p$. Consider the re-labeled graph G in Fig. 4(c) as an example, four subgraphs are constructed, as shown in Fig. 6. Each subgraph$_p$ ($0 \leqslant p \leqslant 3$) consists of a local-edge block, three remote in-edge blocks and three remote out-edge blocks. For each subgraph$_p$, the local-edge block and the three remote in-edge blocks are in the column p; the three remote out-edge blocks are in the row p. Each remote out-edge block includes all the edges whose destination vertices reside within a given remote partition. Each remote in-edge block includes all the edges whose source vertices reside within a given remote partition. The local-edge block B(block$_p$, shard$_p$) is a special case because both the source vertices and destination vertices of the edges are in the partition$_p$.

Compared with graph partitioning process, the time of constructing subgraphs is negligible since each compute node only entails two simple operations: (1) constructing metadata for its subgraph; (2) receiving one in-edge block from other P-1 compute nodes respectively.

Summary: The data representation of LCC-Graph is inspired by GraphChi. The reason is that it is easy to further decompose each shard of GraphChi into edge blocks and then organize the edge blocks into the edge-block based subgraphs of LCC-Graph. However, LCC-Graph is different from GraphChi. GraphChi is a single-node secondary storage based graph processing framework, which organizes the input graph into P shards that reside in the disk of a single compute node, as shown in Fig. 5. By using the Parallel Sliding Window (PSW), GraphChi can execute graph algorithms on the shards [7]. However, LCC-Graph is a pure in-memory distributed graph processing framework, which further decomposes each shard into edge blocks and then organizes the input graph into edge-block based subgraphs, each of which resides in the memory of a compute node, as shown in Fig. 6. This data representation enables the four advantages that significantly reduce the communication costs, as discussed in Section 3. To sum up, although the data representation of LCC-Graph is inspired by GraphChi, LCC-Graph is significantly different from GraphChi because the former is designed to reduce the high communication costs experienced by existing in-memory distributed graph-processing frameworks while the latter aims to improve the performance of single-node disk-based graph-processing frameworks by reducing disk I/O costs.

4.2 Edge data block based communication

Block dependencies: For each subgraph$_p$ ($0 \leqslant p \leqslant P$-1), each remote out-edge block B(block$_p$, shard$_j$) is a copy of the remote in-edge block B(block$_p$, shard$_j$) of subgraph j, where $0 \leqslant j \leqslant P$-1 and $p \neq j$. As shown in Fig. 6, consider subgraph$_0$, the remote out-edge block B(block$_0$, shard$_1$) of subgraph$_0$ is a copy of the remote in-edge block B(block$_0$, shard$_1$) of subgraph$_1$.

Communication: Since only the edge values are mutated during computation, each edge block B(x, y) is split into an adjacency block B_adj(x, y) and an edge data block B_data(x, y). The adjacency blocks store the topological structure of the input graph and the edge data blocks store the edge values. In the communication step, the P-1 remote out-edge data blocks of each compute node will be sent to the other P-1 compute

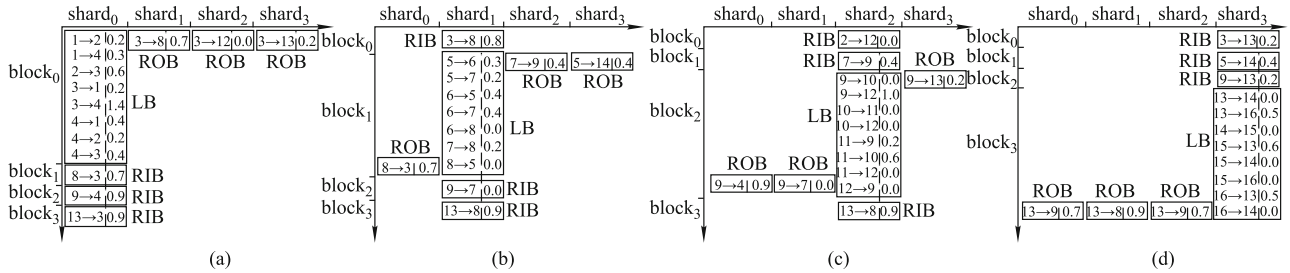


Fig. 6 Subgraphs of G (The *x* axis denotes the shard labels, and the *y* axis denotes the block labels corresponding Fig. 5). (a) Subgraph$_0$; (b) subgraph$_1$; (c) subgraph$_2$; (d) subgraph$_3$

nodes to update their respective copies.

Local edge data block B_data(p, p): The local edge block B(p, p) of each subgraph$_p$ is a special case since both the source and destination vertices of its edges belong to partition$_p$. In the computation step, conflicts can occur when the values of these edges are accessed by the source vertices and destination vertices of these edges simultaneously, which breaks determinism. To guarantee determinism, we implement two copies of the local-edge data block B_data(p, p). One is used for reading and the other for writing. In the next superstep, the two copies switch their roles.

4.3 Network ecosystem

In the network ecosystem, the TCP/IP socket interface calls were initially used for inter-node communication and LCC-Graph was performing over a 1Gbps Ethernet, which can only provide ~110MB/s of actual network bandwidth. Using the edge data block based communication model, LCC-Graph consumes the whole 1Gbps Ethernet network bandwidth, i.e., it can obtain better performance if running on a higher bandwidth network. Hence, we run LCC-Graph on a 40Gbps Infiniband network. However, the TCP/IP protocol now becomes the new bottleneck, due to its low efficiency for multi-layer complex structures. In fact, LCC-Graph can only obtain ~1.2GB/s actual application bandwidth on TCP/IP over the 40Gbps Infiniband network, far below the peak performance. Hence a more efficient communication protocol is required.

The InfiniBand Architecture (IBA) [15] defines a switched network fabric for the interconnection of processing nodes, which provides a communication and management infrastructure for inter-processor communication. In an InfiniBand network, processing nodes are connected to the fabric by channel adapters (CAs). The InfiniBand Architecture supports two semantics, channel and memory. In memory semantics, InfiniBand supports remote direct memory access (RDMA) operations that include RDMA write and RDMA read. RDMA operations are single-sided and do not incur software overheads on the remote side, resulting in efficient communication. Hence, we have implemented LCC-Graph on the InfiniBand Architecture, using RDMA communication operations. In this case, LCC-Graph can obtain ~2.53GB/s actual application bandwidth, further shortening the communication time.

4.4 Remote out-edge data block compression

For most graph algorithms, almost all the edge values are updated in supersteps, such as PargeRank [11], Community Detection [13] and Connected Components [12]. In order to further speed up the graph algorithms with only a few updated edge values, a remote out-edge data block compression (CoDB) scheme is introduced in LCC-Graph.

For each compute node, we define a CoDB for each out-edge data block. Each element in the CoDB includes an *offset* and an edge value, as shown in Fig. 7. The *offset* indicates the edge value offset in the out-edge data block. The edge value is the new value of the edge that is updated in this superstep. Each *offset* is a 4-byte integer number which is large enough to store the maximum value of the offsets due to the limited number of the edge values in each out-edge data block. The extra communication volume for carrying the *offsets* is smaller than that of existing BSP-based distributed graph-processing frameworks since the size of destination vertex names on the messages in the latter is larger than the size of the *offsets*. Like the latter, this scheme also leads to extra overhead at the receiver side. However, by using this scheme, the communication cost of LCC-Graph is still significantly smaller than that of the latter, due to the smaller extra communication volume and higher communication efficiency.

Update counters (UCs): LCC-Graph defines a UC for each remote out-edge data block to record the number of updated edge values in the remote out-edge data block. In the computation step, when an edge is updated, the UC is increased by 1 and a CoDB element will be added to the CoDB. We call this process *compression* that is enabled by a configurable option. Users can disable this option by default for higher performance. Because, for most graph algorithms, al-

most all the edge values are updated in supersteps.

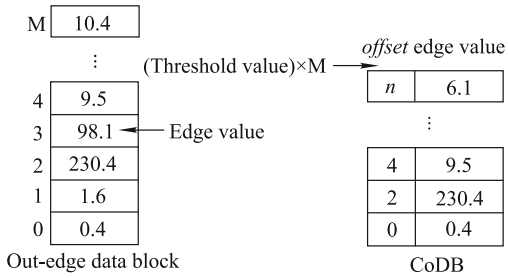


Fig. 7 Remote out-edge data block compression

Threshold value (TV): In the communication step, if the remote out-edge data block compression option is enabled, the CoDBs will be sent to other compute nodes, instead of the full remote out-edge data blocks. However, CoDB introduces additional communication volume to carry the *offset* for each edge value. Hence, a TV is required for each remote out-edge data block to control the ratio of compressed edge values, where $0 \leqslant TV \leqslant 1$. In the compression process, if the ratio of compressed edge values to total edge values of the remote out-edge data block reaches the TV, compression is abandoned. In this case, the remote out-edge data block will be sent in the communication step.

When a compute node has received a CoDB, it uses the *offset* of each element to locate the address of the edge value in the corresponding remote in-edge data block and replace the edge value using the new one.

TV is configurable. Initially, it is set to a value according to the experiences of users. Since the optimal value of TV can be different from one computing environment to another, LCC-Graph can adjust the TV dynamically according to the trends of TV and system performance from the job execution logs.

4.5 Asynchronous execution model

LCC-BSP computation model can fully guarantee determinism by using two synchronous barriers in each superstep. However, asynchronous execution can also guarantee determinism for some graph algorithms, such as Single-Source Shortest-Paths [6] and Label Propagation Algorithm [14]. In this case, asynchronous execution of the LCC-Graph framework can accelerate the convergence of graph-computing jobs, gaining notable performance improvements.

We implement asynchronous execution of LCC-Graph using two measures. First, we relax the determinism requirement by removing the synchronous barrier between the computation step and the communication step in each superstep. After a compute node finishes its computation earlier than other nodes, it sends the P-1 remote out-edge data blocks to other compute nodes immediately. When a compute node receives a remote out-edge data block, it uses the remote out-edge data block to update the corresponding remote in-edge data block directly. Second, for synchronous execution, LCC-Graph uses two copies of the local-edge data block B_data(p, p) to ensure determinism. In asynchronous execution, in contrast, LCC-Graph uses only one copy of local-edge data block B_data(p, p). Thus, in the same superstep, any vertex executed later can read the new in-edge values that have been updated by neighboring vertices that have already finished execution, accelerating the convergence of the graph-computing jobs.

The experimental results, presented in Section 6, show that asynchronous execution improves the overall run time of some graph-computing jobs over synchronous execution by ~29%.

4.6 High flexibility

LCC-Graph also flexibly supports some peculiar requirements of graph algorithms by using a *mignon message block* attached to each out-edge data block, even if these cases rarely occur. These requirements include messages sent to non-neighbor vertices and multiple messages sent to a single destination vertex. Each message field in the *mignon message block* includes *DestinationVertexID* and *MessageValue* fields. Instead of using a global *DestinationVertexID*, LCC-Graph uses a local one for each remote compute node, thus reducing the communication volume. That is, each *mignon message block* is dedicated to one remote compute node independently. Because, due to the process of relabeling vertices, it is easy to identify which compute node the vertices belong to, instead of accessing the global graph-partitioning information. In the computation step, the message fields of each *mignon message block* are updated sequentially in order by using the automatically incremented value of a message field pointer. In the communication step, each *mignon message block* is sent to its remote compute node. Like existing BSP-based distributed graph-processing frameworks, these messages carried by the *mignon message blocks* are separated and routed to their destination vertices by the receiver side, leading to extra overhead at receiver side. However, LCC-Graph is also efficient when executing the graph algorithms with peculiar requirements, due to the reduced communication volume and the efficient network ecosystem. This feature of flexibility is also provided by Pregel [2]. By using this

compensatory technique, LCC-BSP computation model acts exactly as the BSP computation model.

4.7 Fault tolerance

LCC-Graph implements fault tolerance by checkpointing the execution state of compute nodes. In each superstep, vertex-programs update the values of vertices and their out-edges, using the in-edge values and vertex values updated in the previous superstep. Thus, each compute node only needs to checkpoint the vertex values, remote in-edge data blocks and the local-edge data block of its subgraph at the end of the superstep. LCC-Graph uses the checkpointed execution state to execute failure recovery. Due to the checkpointing cost, LCC-Graph requires that the frequency of checkpointing should be fixed (i.e., a set number of supersteps between any two consecutive checkpointing operations), which is a tradeoff between execution delay and recovery time. The default checkpointing frequency is assigned in the system configuration file. Users can disable the fault tolerance mechanism by default for higher performance.

5 Graph partitioning methods

High-quality graph partitioning methods, such as METIS [16], KL [17] and the Genetic algorithm [18], provide a lower edge-cut ratio, resulting in lower communication workloads among compute nodes. However, they fail to handle large-scale graphs in "big data" environments because of their costly computational complexity. By default, existing distributed graph-processing frameworks distribute the vertices of a graph to the compute nodes randomly, due to its simplicity and low computational complexity. This method delivers high partitioning performance but introduces extremely high edge-cut ratios since the locality of the vertices is not considered [19, 20].

In this section, we first present a brief introduction to the heuristic partitioning methods. These methods are useful for partitioning large-scale graphs due to the low computational complexity and the low edge-cut ratios. However, these graph partitioning methods are insufficient for LCC-Graph. Because the communication time in LCC-Graph has been significantly reduced and the computational imbalance among the compute nodes has become a new performance bottleneck. Based on the heuristic partitioning methods, we propose an improved, compute-aware SGP graph partitioning method (CA-SGP) that not only has the low computational complexity and low edge-cut ratios but also balances the computation among the compute nodes, gaining notable reductions in computation time.

5.1 Heuristic partitioning methods

In recent years, lightweight heuristic partitioning methods, such as Streaming Graph Partitioning (SGP) [19] and Restreaming Graph Partitioning (RSGP) [20], have been proposed for large-scale graphs. Compared to the random partitioning method, heuristic partitioning methods can reduce the edge-cut ratios greatly by using a heuristic function to assign each vertex to compute nodes. The heuristic function not only considers the locality of the vertices, but also balances the workload of each compute nodes. In SGP, vertices along with their neighbors of the input graph arrive in order. SGP uses a user-defined heuristic function to assign the arriving vertices to different partitions. It uses the following heuristic to select a partition $\boldsymbol{i}$ for a vertex $\boldsymbol{u}$ by default:

$$\underset{i\in\{1,\ldots,K\}}{argmax}|P_i^t \cap N(u)|(1-\frac{WL_i^t}{EWL_i}), \tag{1}$$

where K is the number of partitions, with each partition being managed by a single compute node. $N(u)$ is the set of u's neighbors and P_i^t is the set of vertices in partition $\boldsymbol{i}$ at time t. The factor $|P_i^t \cap N(u)|$ is used to reduce the edge-cut ratio. The factor $1-\frac{WL_i^t}{EWL_i}$ is a compensation factor to balance the workloads of the compute nodes, where WL_i^t is the workload of the compute node $\boldsymbol{i}$ at time t, and EWL_i is defined as the expected workload of compute node i. The SGP partitioning method usually defines a workload unit as an "edge". Hence, if the cluster is built of homogeneous/identical compute nodes, EWL_i can be defined as NE/K, where NE is the number of edges of the input graph. WL_i^t is the number of edges of compute node $\boldsymbol{i}$ at time t.

The experimental results, as shown in Section 6, indicate that if the workload unit is defined as an "edge", the SGP partitioning method can obtain an ideal communication balance, due to the similarity among the partitions in terms of the number of edges. The communication time required by each compute node depends on the number of cut-edges, which is proportional to the product of the edge-cut ratio and the number of edges of each partition.

However, this heuristic function introduces huge differences in the number of vertices in each partition, resulting in a serious computational imbalance. The reason for this is that the computation time for most graph algorithms depends not only on the number of edges, but also is slightly dependent on the number of the vertices of the partition, since scheduling the vertices is time-consuming. The large vertex imbalance

results from the non-uniform distributions of the "celebrity" vertices that have the most edges in the graphs.

5.2 Improved, compute-aware SGP (CA-SGP)

The SGP partitioning only balances the communication workload among compute nodes. It is sufficient for existing distributed graph-processing frameworks, since the communication costs of them are the main contributors to the overall system performances. However, the SGP partitioning is insufficient for LCC-Graph since the communication time is very short. Hence, the computation load-balance problem becomes the new bottleneck for LCC-Graph, which must be addressed to further improve the system performance. To end this, we propose to optimize the heuristic function to mitigate the computation imbalance, making it compute-workload aware.

Like SGP, vertices along with their neighbors of the input graph arrive in order. CA-SGP also uses the heuristic function of SGP to assign the arriving vertices to different partitions, as shown in Eq. (1). However, based on the SGP, CA-SGP further balances the computation workload among the compute nodes by optimizing the factor $1 - \frac{WL_i^t}{EWL_i}$ of Eq. (1). Specifically, we redefine EWL_i as:

$$EWL_i = EE_i \times \alpha + EV_i \times (1 - \alpha), \tag{2}$$

where α is a balance factor which balances the workload impacts of edges and vertices, $0 \leqslant \alpha \leqslant 1$. EE_i is defined as the number of edges that compute node i expects to accept. If the cluster is built of homogeneous/identical compute nodes, EE_i can be defined as NE/K, where NE is the number of edges of the input graph. Similarly, EV_i is defined as the number of vertices that compute node i expects to accept. If the cluster is built of homogeneous/identical compute nodes, EV_i can be defined as NV/K, where NV is the number of vertices of the input graph. WL_i^t is defined as:

$$WL_i^t = NE_i^t \times \alpha + NV_i^t \times (1 - \alpha), \tag{3}$$

where NE_i^t is the number of edges in partition $\boldsymbol{i}$ at time t, and NV_i^t is the number of vertices in partition $\boldsymbol{i}$ at time t.

An appropriate value of α slightly increases the communication imbalance, but greatly mitigates the computation imbalance, resulting in a higher performance of LCC-Graph. α is a configurable parameter that can be assigned. The experimental results presented in Section 6 indicate that the improved, compute-aware SGP (CA-SGP), exhibits notable performance improvements for LCC-Graph when the balance factor is set to 0.85.

Summary: The computation imbalance problem becomes the new bottleneck for LCC-Graph since the communication time has been significantly reduced by our LCC-BSP computation model. By using the balance factor α, CA-SGP graph partitioning method balances the workload impacts of edges and vertices, gaining notable reductions in computation time for each superstep. The experimental results, as shown in Section 6, indicate that CA-SGP balances not only the computation workloads but also the communication workloads, resulting in higher system performance of LCC-Graph, compared with the original SGP.

6 Experimental evaluation

In this section, we conduct extensive experiments to evaluate the performance of LCC-Graph. Experiments are conducted on a 32-node cluster. Each compute node has two quad-core Intel Xeon E5620 processors with 24GB of RAM. Nodes are connected via a 40Gbps InfiniBand network and a 1Gpbs Ethernet for high-bandwidth and low-bandwidth interconnect evaluations respectively.

Graph algorithms: We implement several graph algorithms to evaluate LCC-Graph: Single-Source Shortest-Paths (SSSP) [6], PageRank (PR) [11], Community Detection (CD) [13] and Connected Components (CC) [12]. The SSSP algorithm computes the distance of the shortest path from a given source vertex u to each other vertex in a graph. The PR algorithm is used by Google Search to rank websites in their search engine. The CC algorithm finds connected components of a given graph, i.e., a maximum set of vertices in which any pair of vertices can reach each other. The CD algorithm works as follows. Each vertex is initially assigned a unique label. Each vertex updates its label with the label most frequently used by its neighbors. This process is repeated until a stable set of labels is reached. We define the sets of vertices that have the same labels as "network communities".

Baseline frameworks: We compare LCC-Graph with GPS, Giraph and GraphLab. GPS is an open-source Pregel implementation from Stanford's InfoLab [3]. It is a representative BSP-based distributed graph-processing framework. GraphLab is an open-source project originated at CMU [21] and now supported by GraphLab Inc. It has competitive performance with the existing distributed graph-processing frameworks. We use the latest version of GraphLab 2.2 (released in March 2014), which supports distributed computation and incorporates the features and improvements of PowerGraph [21,22]. We also choose Apache Giraph [4] as it is a

well-known open source implementation of Pregel.

Datasets: We evaluate LCC-Graph using several real-world graph datasets that are summarized in Table 1. These datasets are used frequently in many published comparative evaluations of graph-processing frameworks [3, 7, 22].

Preprocessing: We conduct experiments to evaluate GPS, GraphLab and LCC-Graph in terms of preprocessing time. Each framework runs two graph partitioning methods of SGP and CA-SGP on Twitter-2010. As shown in Fig. 8, the preprocessing time of LCC-Graph is 1.12x longer than that of GPS but 1.19x shorter than that of GraphLab when the same graph partitioning method is used. We then decompose the preprocessing runtime of LCC-Graph into four parts: 1) graph partitioning time 2) relabeling time 3) sharding time and 4) the time of constructing subgraphs. The experimental results indicate that the preprocessing time of LCC-Graph is dominated by the graph partitioning time and the sharding time. They are responsible for 71.78% and 27.61% of the overall preprocessing time respectively. The preprocessing times are not included in calculations in the other experiments described in the following subsections.

Table 1 Graph datasets summary

Datasets	$\|V\|$	$\|E\|$	Type
LiveJournal [23]	4.8×10^6	69×10^6	Social Network
Twitter-2010	41×10^6	1.4×10^9	Social Network
UK-2007-05	106×10^6	3.7×10^9	Web
USA	23.9×10^6	58.4×10^6	Road Network
Euro	18×10^6	44.8×10^6	Road Network

6.1 Runtime breakdown

Experiments are conducted to investigate LCC-Graph and GPS in terms of communication cost and computation time. GraphLab is excluded from this evaluation since it is difficult to explicitly measure its communication cost due to its distributed shared-memory technique. Each framework runs the PR, CD, CC and SSSP graph algorithms on 24 compute nodes with the Twitter-2010 graph. Compute nodes are connected via a 40Gbps InfiniBand network. PR runs 10 supersteps for each experiment.

Communication cost: The communication cost is defined as the time spent by vertices to interact with one another. In order to measure the communication cost and computation time more easily, each superstep of LCC-Graph is explicitly divided into a computation step and a communication step by using a global barrier. Thus, in each superstep, the communication cost is the time for the communication step. For GPS, the communication cost consists of the sender-side communication overhead, the time spent on sending message batches, and the receiver-side communication overhead.

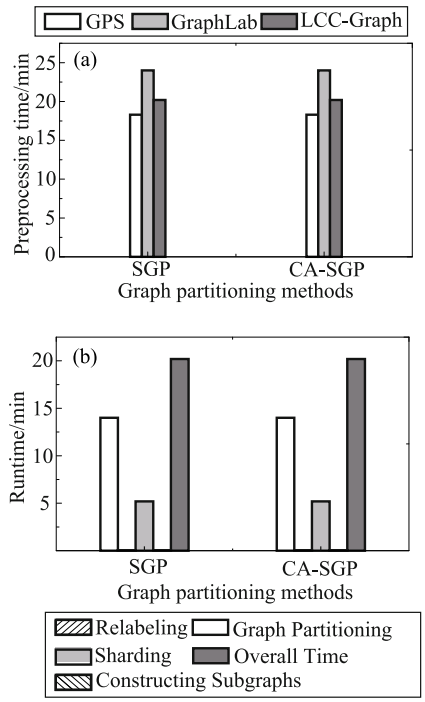


Fig. 8 Preprocessing time. (a) Overall runtime; (b) runtime breakdown

For fair comparison, GPS is evaluated with two configurations of message buffer size, i.e., 100KB (the default value) and 80MB which is large enough to accommodate all the messages sent from one compute node to another in each superstep. We record the communication cost for each experiment. Figure 9(a) shows the difference between LCC-Graph and GPS in terms of communication cost. GPS with the 80MB buffer configuration only gains less than 6% improvements in the communication cost, compared to its default configuration. The reasons are the high extra volume of communication, limited effectiveness of the *message buffering* technique and poor communication bandwidth utilization in GPS, as discussed in Section 2. However, the communication cost of LCC-Graph is 59x, 60x, 66x, and 14x shorter than that of GPS with the 80MB buffer configuration when running the PR, CD, CC and SSSP algorithms respectively. The reasons are the reduced communication volume, the highly efficient communication and the elimination of the extra overhead at the receiver side. We also observe that the communication cost gap between LCC-Graph and GPS is narrower for SSSP than that for the other graph algorithms. The reason is that the SSSP graph algorithm generates only a few inter-vertex interactions during the execution process. This restricts the advantage of low communication costs in LCC-

Graph. Even so, the communication cost of LCC-Graph is still 14x shorter than that of GPS.

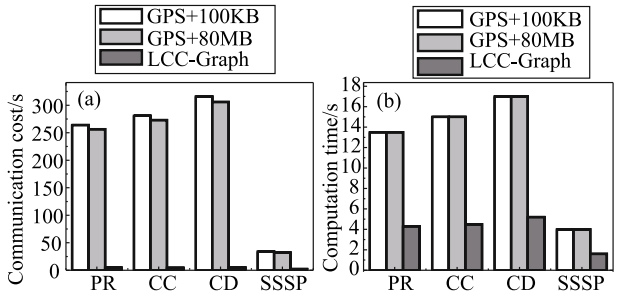


Fig. 9 Runtime breakdown. (a) Communication cost; (b) computation time

The experimental results indicate that the communication cost of GPS dominates the overall run time in each graph-computing job. For example, when GPS performs the PageRank algorithm, the communication cost is responsible for 95% of the overall run time. On the contrary, it is the low communication cost of LCC-Graph that contributes to its high performance in each graph-computing job.

Computation Time: In these experiments, we also measure the time spent on computation in each graph-computing job. Experimental results, as shown in Fig. 9(b), indicate that the computation time of LCC-Graph is 2.5x-3.4x shorter than that of GPS. As discussed in Section 3, this performance improvement stems from the reduced computation work of each vertex. The reduced computation time is another contributor to the high performance of LCC-Graph.

6.2 Impacts of network ecosystem

Experiments are also conducted to study the impact of the network ecosystem on the performance of these frameworks. In these experiments, each framework is deployed on 16 compute nodes, and runs 10 supersteps of PR on the Twitter-2010 graph.

Over the 1Gbps ethernet: We first test LCC-Graph against GraphLab and GPS over the 1Gbps Ethernet. As illustrated in Fig. 10(a), LCC-Graph is 5.6x and 1.7x faster than GPS and GraphLab respectively. The run-times of LCC-Graph are similar when running on either TCP/IP or RDMA. The reason is that edge data block based communication model saturates the 1Gbps network bandwidth when running on either TCP/IP or RDMA. In fact, the actual obtained bandwidth, calculated as the size of the out-edge data blocks divided by the communication time, is ~110MB/s, which reaches the upper limit of the 1Gbps Ethernet network. In these experiments, the network bandwidth is the performance bottleneck in LCC-Graph.

Over the 40Gbps infiniband: We then evaluate LCC-Graph against GraphLab and GPS over the 40Gbps Infiniband. The experimental results shown in Fig. 10(a) indicate that LCC-Graph with RDMA is 1.4x faster than LCC-Graph with TCP/IP, due to higher efficiency of RDMA, and is 41x and 11x faster than GPS and GraphLab respectively. Compared with the 1Gbps Ethernet, LCC-Graph, when running over the 40Gbps Infiniband, obtains significant performance improvements, which is not the case for GPS and GraphLab that fail to observe any significant performance improvements. The experimental results indicate that LCC-Graph has significantly higher efficiency when the system is supported by a high-quality network ecosystem.

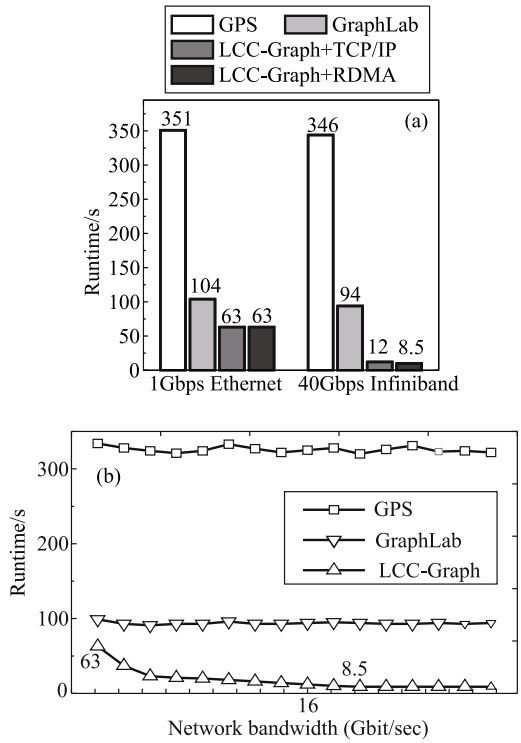


Fig. 10 Impacts of network ecosystem. (a) Different networks; (b) different network bandwidth

Different network bandwidth: We conduct experiments to evaluate the scalability of these frameworks in terms of network bandwidth. Each experiment is repeated by gradually increasing the network bandwidth from 1Gbps to 40Gbps. As illustrated in Fig. 10(b), LCC-Graph, running on RDMA, achieves the peak performance when the network bandwidth is limited to 20Gbps. Consider the 1Gbps-based configuration as the baseline, the peak performance (8.5s) is 7.4x higher than the baseline performance (63s). When LCC-Graph achieves its peak performance, the measured actual obtained bandwidth is ~2.53 GB/s. However, a higher net-

work bandwidth does not contribute to higher performances of GraphLab and GPS.

6.3 Out-edge data block compression (CoDB)

We also study the CoDB that can help speedup the graph-computing jobs with a few inter-vertex interations. LCC-Graph runs SSSP on 24 compute nodes with the Twitter-2010 graph, ranging the TV (threshold value) from 0 to 0.5. In fact, CoDB is disabled when TV=0. We regard this case as the baseline. As shown in Fig. 11, the runtime decreases gradually and reaches a minimum value when TV=0.08. The reason is that a larger TV value enables more out-edge data blocks to be compressed, gaining a reduction in communication time. The runtimes maintain the minimum value when TV ranges from 0.08 to 0.5. The reason is that the maximally connected component of the Twitter graph only has ~3 million vertices while the Twitter graph has ~41 million vertices. The ratio of the scheduled vertices is less than 7.2%, leading to a small proportion of edges being updated during the execution process. Hence, most out-edge data blocks can be compressed when TV is larger than 0.08. Overall, CoDB is able to reduce the runtime of SSSP by 26%.

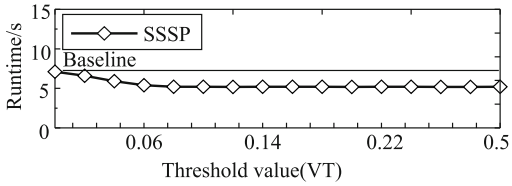


Fig. 11 Effectiveness of CoDB

6.4 Impacts of graph partitioning methods

The CA-SGP graph partitioning method proposed in Section 5 will now be studied, and we will compare CA-SGP with the basic SGP. In order to measure the communication time and computation time more easily, we also uses a global barrier between the computation step and the communication step in each superstep.

SGP: We conduct experiments to evaluate the SGP graph partitioning method. In this experiment, LCC-Graph, deployed on 8 compute nodes, runs 10 supersteps of PR on the Twitter-2010 graph. As illustrated in Fig. 12(c), there is very little difference among the communication times of all compute nodes. The reason is that each subgraph has a nearly equal number of edges, as shown in Fig. 12(a), since SGP defines the number of edges as the balancing objective, which leads to communication balance. However, the differences among the computation times of the compute nodes are large, for example, the computation time for compute node 0 is 3x longer than that for compute node 2. The reason is that SGP only balances the number of edges of each subgraph, leading to a serious vertex imbalance. As shown in Fig .12(b), the number of vertices of Subgraph 0 is 30x larger than that of Subgraph 2. The serious vertex imbalance results in an imbalance in the computation workloads among compute nodes.

Improved, Compute-Aware SGP (CA-SGP): As discussed in Section 5, the heuristic function has been optimized to mitigate the computational imbalance of SGP. We use the balancing factor α to balance the workload impacts of edges and vertices. A series of experiments have been conducted to investigate the optimal value of α. The experimental results indicate that LCC-Graph can reach optimal performance when α=0.85. Due to space limitations, we only show the experimental results with α=0.85. CA-SGP brings a slightly higher edge imbalance (as shown in Fig. 13(a)), compared to SGP, but results in better vertex balance (as shown in Fig. 13(b)). Hence, it obtains ~30% performance improvement over SGP, as illustrated in Fig. 13(c).

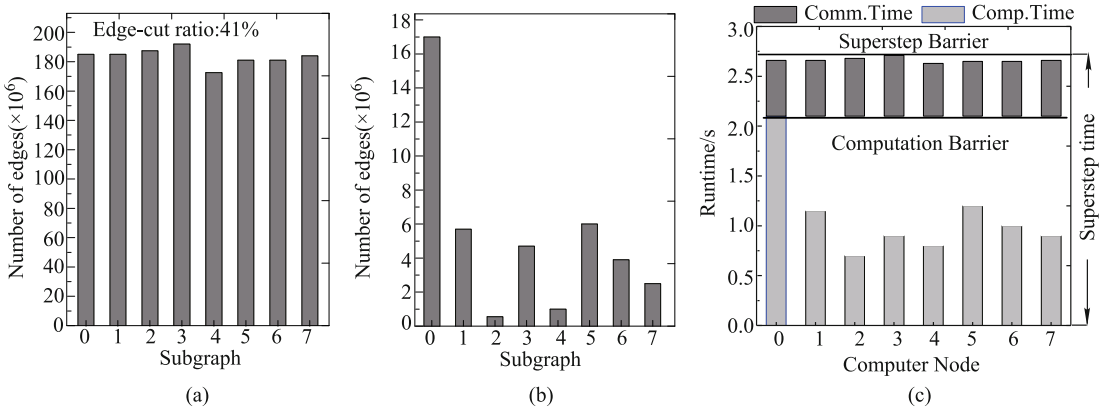


Fig. 12 SGP partitioning method. (a) Edge balance; (b) vertex balance; (c) runtime

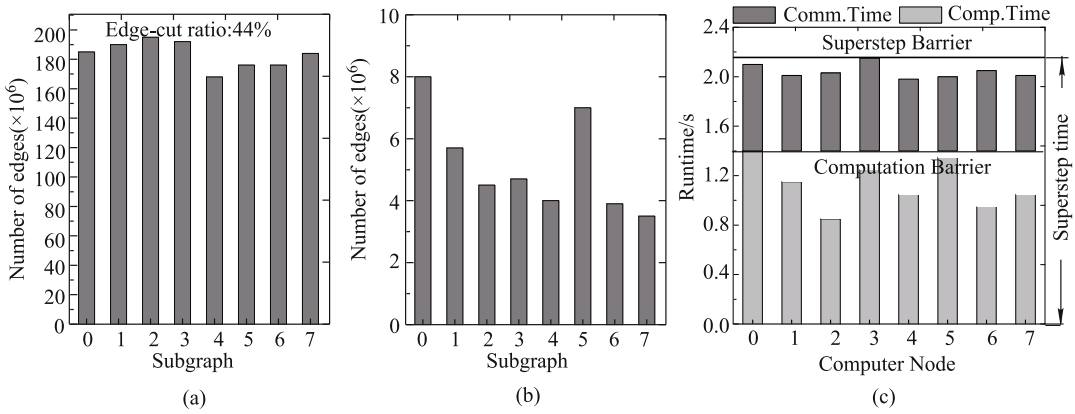


Fig. 13 CA-SGP partitioning method. (a) Edge balance; (b) vertex balance; (c) runtime

6.5 Asynchronous execution model

We conduct experiments to evaluate the performance of LCC-Graph with asynchronous execution. LCC-Graph runs SSSP and PR over the Twitter-2010 graph on 24 compute nodes with synchronous and asynchronous execution models respectively. Experimental results, as shown in Fig. 14, indicate that LCC-Graph can accelerate the convergence of many graph algorithms by gaining notable reductions in the number of supersteps, which contribute to the shorter run times. For example, the SSSP graph-algorithm job achieves convergence by the 12th superstep in asynchronous execution. However, convergence is not achieved until the 17th superstep in synchronous execution. In this experiment, the performance improvement is ∼29%. The reason behind the faster convergence of the asynchronous mode is that, in the same superstep, any vertex executed later can read the new in-edge values that have been updated by neighboring vertices that have already finished execution.

6.6 Comparison with existing systems

We evaluate LCC-Graph comprehensively against GPS, Giraph and GraphLab. Each framework runs four graph algorithms on various graph datasets (as shown in Table 1).

Experimental results, as shown in Figs. 15(a)–(c), indicate that the speedup of LCC-Graph is higher when running on bigger graph datasets. For example, LCC-Graph is 27.3x, 44.1x and 49.6x faster than GPS respectively when running PR with 10 supersteps on the LiveJournal, Twitter and UK-2007-05 graphs. The reason for this is the higher scalability of LCC-Graph, since more compute nodes required by bigger graph dataset bring higher speedup. In these experiments, LCC-Graph runs 12x–49.6x faster than GPS on various graph algorithms and graph datasets. Giraph is a well-known open source implementation of Pregel, which is being actively developed in recent years. In our comprehensive evaluation, Giraph obtains ∼7% performance improvement over GPS when running CC with the Twitter graph. In other cases, GPS runs 1.04x–1.17x faster than Giraph. In our comprehensive evaluation, LCC-Graph runs 6.7x–14.5x faster than GraphLab on various graph algorithms and graph datasets. We also notice that the performance improvement of LCC-Graph is reduced when executing SSSP, compared with other graph algorithms (PR, CC and CD). This is because SSSP requires a few interactions among vertices. This restricts the advantage of low communication costs in LCC-Graph.

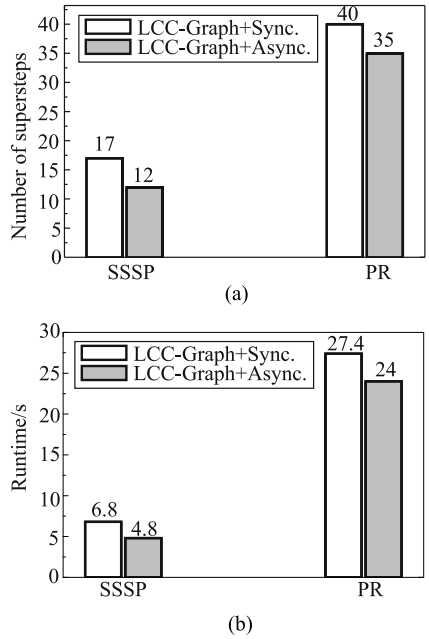


Fig. 14 Asynchronous execution model. (a) Supersteps; (b) runtime

LCC-BSP also supports the peculiar requirement of sending messages to non-neighbor vertices by using a *mignon message block* attached to each out-edge data block, even if some graph processing frameworks do not provide this

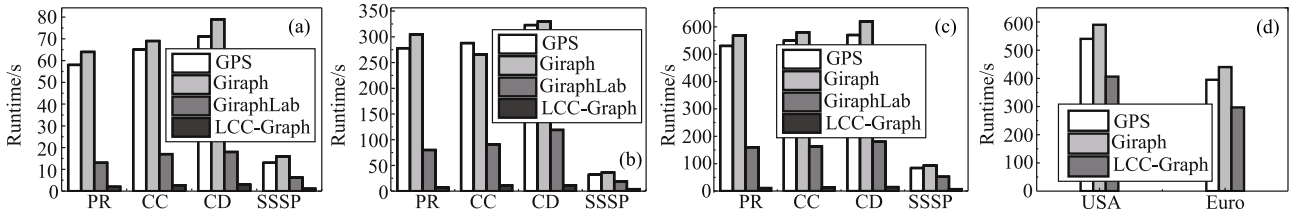


Fig. 15 Comparison with existing systems. (a) LiveJournal + 3nodes; (b) Twitter-2010 + 24nodes; (c) UK-2007-05 + 32nodes; (d) BPPA for List Ranking + 8nodes

feature [24]. Experiments are conducted to investigate LCC-Graph, GPS and Giraph in terms of this feature. GraphLab is excluded from this evaluation since it can not send messages to non-neighbor vertices. Each framework runs *BPPA List Ranking* on 8 compute nodes with the *USA* graph. Compute nodes are connected via a 40Gbps InfiniBand network. *BPPA List Ranking* is a graph algorithm that is designed by Da Yan et al. for list ranking, in which each vertex interacts with non-neighbor vertices in each superstep after the first one [24]. The *USA* graph is preprocessed to obtain a linked list ζ for *BPPA List Ranking*. For a linked list ζ, each vertex, except the first one, only has a outgoing edge to its predecessor. As shown in Fig. 15(d), LCC-Graph obtains 33% and 45.3% performance improvements over GPS and Giraph respectively. Experiment is also conducted on *Euro* graph, and get the similar results. Experimental results indicate that LCC-Graph is also effective for the requirement of sending messages to non-neighbor vertices, in spite of the reduced performance improvement, compared with the case of sending messages to neighbor vertices.

7 Related work

Pregel [2] adopts a combiner to reduce the number of cross-machine messages. However, due to poor spatial locality among the destination vertices, only a relatively small number of messages can be combined. Furthermore, these solutions introduce extra overheads. Finally, a combiner may not be useful in many graph algorithms where the values of the messages are not commutative or combinative. GiraphUC [25] adopts a barrierless asynchronous parallel (BAP) computation model to reduce both message staleness and global synchronization. Another alternative solution to reduce cross-machine messages is using advanced graph partitioning strategies [3, 19, 22, 26] to lower the number of cut-edges across nodes. These strategies are also useful for LCC-Graph. Several systems, such as GraphX [27] and PowerGraph [22], can reduce communication cost by partitioning vertices instead of edges among subgraphs to evenly distribute edges of high-degree vertices, but they also incur high communication cost among partitioned low-degree vertices.

Giraph [4] serializes the edges and messages into byte arrays to reduce the number of objects, aiming to improve the memory utilization. Furthermore, a superstep splitting technique is developed to split a message-heavy superstep into several steps, so that the number of messages transmitted in each step does not exceed the memory size [28]. Pregel+ develops two techniques to reduce the number of messages. The first technique is to create mirrors of each high-degree vertex, aiming to combine the messages of the high-degree vertex. However, since a mirrored vertex forwards its value directly to its mirrors, it loses the chance of message combining. Therefore, there is a tradeoff between vertex mirroring and message combining in reducing the number of messages [28]. The second technique is designed for pointer jumping algorithms where a vertex needs to communicate with a large number of other vertices that may not be its neighbors. This technique can prevent vertex r from receiving and sending a lot of messages, by combining all requests on each worker as one request towards vertex r [28]. GPS [3] introduces the dynamic repartitioning and large adjacency list partitioning (LALP) techniques to reduce the number of messages sent over the network. However, dynamic repartitioning also introduces extra network workload to reassign vertices among workers, leading to overhead that sometimes exceeds the benefits gained [29]. MOCgraph [30] adopts the message online computing (MOC) model to eliminate the memory space consumed by messages. However, LCC-Graph reduces the communication cost by eliminating the high extra volume of communication, avoiding the data copying and batch parsing overheads, and by improving the communication bandwidth utilization.

Some distributed graph-processing systems, such as Pregel, adopt a combiner at the sender side to reduce the number of cross-machine messages [2]. However, some distributed graph-processing systems, such as GPS, do not support sender-side combining due to the reasons as follows [3]. First, due to poor spatial locality among the destination vertices, only a relatively small number of messages can be

combined. Second, this solution introduces extra overheads. Third, a combiner may not be useful in many graph algorithms where the values of the messages are not commutative or combinative. Like GPS, LCC-Graph do not implement the message combiner due to the reasons above.

Blogel [31] is a block-centric framework which uses parallel algorithms to partition an arbitrary graph into blocks efficiently, and block-centric programs are then run over these blocks. This framework is significantly faster than existing distributed graph-computing systems, for processing large graphs with adverse graph characteristics such as skewed degree distribution, high average degree, and large diameter. However, LCC-Graph is designed to reduce the high communication costs by using the edge-data-block data representation. Yan et al. [32] propose a new open-source system, called Quegel, for querying big graphs. Quegel is highly efficient in answering various types of graph queries and is up to orders of magnitude faster than existing systems. Like Google's Pregel [2], LCC-Graph is designed mainly for large-scale graph analytics. However, by using the remote out-edge data block compression (CoDB) scheme, LCC-Graph can also help speedup the graph-computing jobs with a few inter-vertex interations.

ParMETIS [33] is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs. The algorithms in ParMETIS are based on the multilevel partitioning algorithms that are implemented in the widely-used serial package METIS [23]. ParMETIS dramatically reduces the time spent on communication by computing mesh decompositions so that the number of interface elements is minimized. LogGP [34] is a log-based graph partitioning system that records, analyzes and reuses the historical statistical information to refine the partitioning result. GPS [3] introduces the dynamic repartitioning and large adjacency list partitioning (LALP) techniques to reduce the number of messages sent over the network. These graph partitioning methods are very useful for existing distributed graph-processing frameworks, since the communication costs of them are the main contributors to the overall system performances. However, these methods are insufficient for LCC-Graph since the communication time is very short and the computation load-balance problem becomes the new bottleneck for LCC-Graph. LCC-Graph addresses this problem by using the proposed CA-SGP partitioning method, gaining significant performance improvements.

In order to reduce hardware costs, several single-node secondary storage based graph-processing frameworks have been proposed to handle graphs with billions of edges [7, 19]. However, the performance of these frameworks is limited by the limited secondary storage bandwidth of a single compute node [35] and the significant difference in the access speeds between secondary storage and main-memory [36]. Furthermore, the limited amount of storage of a single commodity computer can potentially limit the scale of the processed graphs, since graphs continue to grow in size [37]. GraphChi [7] introduces a novel mechanism called Parallel Sliding Windows (PSW) to alleviate the issue of random accesses to improve the disk I/O performance. X-Stream [19] avoids the random accesses by streaming completely unordered edge lists, aiming to improve the disk I/O performance. GridGraph [38] also uses edge-centric computing model. Through a novel dual sliding windows method, GridGraph can stream the edges and apply on-the-fly vertex updates, thus reduce the I/O amount required for computation. VENUS [39] separates read-only structure data from mutable vertex data on disk, significantly reducing random I/O amount.

Chaos [37] is disk-based distributed graph processing framework, which scales X-Stream [19] out to multiple machines. There are two key challenges in the design of a distributed secondary storage based system, that is, the expensive communication and the high disk I/O latency. Chaos reduces the disk I/O latency by streaming completely unordered edge lists. However, Chaos' system performance relies heavily on the assumption that network bandwidth far outstrips storage bandwidth [37]. The reason is that, in order to achieve load balance at runtime, Chaos employs multiple machines to execute a single partition. Hence, Chaos employs RDMA (remote direct memory access) technique to provide high network bandwidth, aiming to reduce communication time. GraM [40] improves communication efficiency by using the RDMA-based communication stack that preserves parallelism in a balanced way and allows overlapping of communication and computation. LCC-Graph also utilizes RDMA to speed up the transfers of the out-edge data blocks. However, the high performance of LCC-Graph relies heavily on its edge-block based data representation. This data representation avoids the extra communication overheads in both the sender side and the receiver side, and significantly reduces the communication volume. Hence, LCC-Graph is also efficient in low-bandwidth networks.

8 Conclusion

This paper proposes a distributed graph-processing frame-

work, called LCC-Graph, to support large-scale graph-computing jobs. LCC-Graph is high-performance and highly scalable while maintaining the advantages of Pregel-like distributed graph-processing frameworks. In LCC-Graph, the LCC-BSP computation model is proposed to eliminate the high communication costs that affect existing distributed graph-processing frameworks, and reduce the computation workload of each vertex. Extensive prototype evaluation of LCC-Graph, driven by real-world datasets, indicates that the performance of LCC-Graph is notably superior to the existing distributed graph-processing frameworks. For example, it runs ~49x and ~14x faster than GPS and GraphLab respectively.

Acknowledgements This work is supported by NSFC 61772216, Shenzhen science and technology plan project (JCYJ 20170307172248636), Wuhan application basic research project (2017010201010103). This work is also supported by Key Laboratory of Information Storage System, Ministry of Education and State Key Laboratory of Computer Architecture (CARCH201505).

References

1. Valiant L G. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8): 103–111
2. Malewicz G, Austern M H, Bik A J C. Pregel: a system for large-scale graph processing. In: Proceedings of ACM International Conference on Management of Data. 2010, 135–146
3. Salihoglu S, Widom J. GPS: a graph processing system. In: Proceedings of ACM International Conference on Scientific and Statistical Database Management. 2013, 22–32
4. Ching A, Edunov S, Kabiljo K, Logothetis D, Muthukrishnan S. One trillion edges: graph processing at facebook-scale. Proceedings of the VIDB Endowment, 2015, 8(12): 1804–1815
5. Wang G, Xie W, Demers A J, Gehrke J. Asynchronous large-scale graph processing made easy. in: Proceedings of International Conference on Innovation Database Research. 2013, 135–146
6. Simmhan Y, Kumbhare A, Wickramaarachchi C, Nagarkar S, Ravi S, Raghavendra C, Prasanna V. Goffish: a sub-graph centric framework for large-scale graph analytics. In: Proceedings of Euro-Par Parallel Processing. 2014, 451–462
7. Kyrola A, Blelloch G E, Guestrin G. Graphchi: large-scale graph computation on just a PC. In: Proceedings of Usenix Conference on Operating Systems Design and Implementation. 2012, 31–46
8. Cheng Y L, Wang F, Jiang H, Hua Y, Feng D, Wang X N. DD-graph: a highly cost-effective distributed disk-based graph-processing framework. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing. 2016, 259–262
9. Cheng Y L, Wang F, Jiang H, Hua Y, Feng D, Wang X N. LCC-graph: a high-performance graph-processing framework with low communication costs. In: Proceedings of IEEE/ACM International Symposium on Quality of Service. 2016, 91–100
10. Cheng Y L, Jiang H, Wang F, Hua Y, Feng D. BlitzG: exploiting high-bandwidth networks for fast graph processing. In: Proceedings of IEEE International Conference on Computer Communications. 2017, 2340–2348
11. Page L. The pagerank citation ranking : bringing order to the web. Stanford Digital Libraries Working Paper, 1998, 9(1): 1–14
12. Stefano L D, Bulgarelli A. A simple and efficient connected components labeling algorithm. In: Proceedings of International Conference on Image Analysis and Processing. 1999, 322–327
13. Fortunato S. Community detection in graphs. Physics Reports, 2010, 486(3): 75–174
14. Kothari R, Jain V. Learning from labeled and unlabeled data. In: Proceedings of IEEE International Joint Conference on Neural Networks. 2002, 2803–2808
15. Lee M, Kim E J, Yousif M. Security enhancement in infiniband architecture. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium. 2005, 105–114
16. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. Journal of Scientific Computing, 1998, 20(1): 359–392
17. Kernighan B W, Lin S. An efficient heuristic procedure for partitioning graphs. Journal of Bell System Technical, 1970, 49(2): 291–307
18. Bui T N, Moon B R. Genetic algorithm and graph partitioning. IEEE Transactions on Computers, 1996, 45(7): 841–855
19. Roy A, Mihailovic I, Zwaenepoel W. X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. 2013, 472–488
20. Nishimura J, Ugander J. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In: Proceedings of ACM International Conference on Knowledge Discovery and Data Mining. 2013, 1106–1114
21. Low Y, Bickson D, Gonzalez J E, Guestrin C, Kyrola A. Distributed graphlab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment, 2012, 5(8): 716–727
22. Gonzalez J E, Low Y, Haijie G, Danny B, Carlos G. Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of Usenix Conference on Operating Systems Design and Implementation. 2012, 17–30
23. Backstrom L, Huttenlocher D, Kleinberg J, Lan X. Group formation in large social networks: membership, growth, and evolution. In: Proceedings of the 12th ACM International Conference on Knowledge Discovery and Data Mining. 2006, 44–54
24. Yan D, Cheng J, Xing K, Lu Y, Ng W, Bu Y G. Pregel algorithms for graph connectivity problems with performance guarantees. Proceedings of the VLDB Endowment, 2014, 7(14): 1821–1832
25. Han M, Daudjee K. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. Proceedings of the VLDB Endowment, 2015, 8(9): 950–961
26. Yang S, Yan X, Zong B, Khan A. Towards effective partition management for large graphs. In: Proceedings of ACM Conference on Management of Data. 2012: 517–528
27. Xin R S, Crankshaw D, Dave A, Gonzalez J E, Franklin M J, Stoica I. Graphx: unifying data-parallel and graph-parallel analytics, Computer

Science, 2014, 8(3): 125–137

28. Da Y, Yingyi B, Yuanyuan T, Deshpande A. Big graph analytics platforms. Foundations and Trends in Databases, 2017, 7(2): 180–195
29. Lu Y, Cheng J, Yan D, Wu H. Large-scale distributed graph computing systems: an experimental evaluation. Proceedings of the VLDB Endowment, 2014, 8(3): 281–292
30. Zhou C, Gao J, Sun B, Yu J X. Mocgraph: scalable distributed graph processing using message online computing. Proceedings of the VLDB Endowment, 2014, 8(4): 377–388
31. Yan D, Cheng J, Lu Y, Ng Y. Blogel: a block-centric framework for distributed computation on real-world graphs. Proceedings of the VLDB Endowment, 2014, 7(14): 1981–1992
32. Yan D, Cheng J, Ozsu M T, Lu Y. A general-purpose query-centric framework for querying big graphs. Proceedings of the VLDB Endowment, 2016, 9(7): 564–575
33. Devine K D, Boman E G, Heaphy R T. New challenges in dynamic load balancing. Applied Numerical Mathematics, 2005, 52(2): 133–152
34. Cheng J, Liu Q, Li Z, Fan W, Lui J C S. VENUS: vertex-centric streamlined graph computation on a single PC. In: Proceedings of IEEE International Conference on Data Engineering. 2015, 1131–1142
35. Malicevic J, Roy A, Zwaenepoel W. Scale-up graph processing in the cloud:challenges and solutions. In: Proceedings of International Workshop on Cloud Data and Platforms. 2014, 1–6
36. Pearce R, Gokhale M, Amato N M. Scaling techniques for massive scale-free graphs in distributed (external) memory. In: Proceedings of International Symposium on Parallel and Distributed Processing. 2013, 825–836
37. Roy A, Bindschaedler L, Malicevic J, Zwaenepoel W. Chaos: scale-out graph processing from secondary storage. In: Proceedings of Symposium on Operating Systems Principles. 2015, 410–424
38. Zhu X, Han W, Chen W. GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of Usenix Conference on Usenix Technical Conference. 2015, 375–386
39. Xu N, Chen L, Cui B. LogGP: a log-based dynamic graph partitioning method. Proceedings of the VLDB Endowment, 2014, 7(14): 1917–1928
40. Ming W, Fan Y, Jilong X, Xiao W, Miao Y. GRAM: scaling graph computation to the trillions. In: Proceedings of ACM Symposium on Cloud Computing. 2015, 408–421

Yongli Cheng received the BE degree from the Chang'an University, China in 1998, and the MS degree from the FuZhou University, China in 2010. He is currently a PhD student majoring in computer architecture in Huazhong University of Science and Technology, China. His current research interests include computer architecture and graph computing. He has several publications in international conferences, including HPDC and IWQoS.

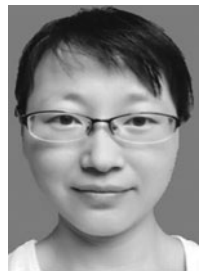
Fang Wang received her BE degree and Master degree in computer science in 1994, 1997, and PhD degree in computer architecture in 2001 from Huazhong University of Science and Technology (HUST), China. She is a professor of computer science and engineering at HUST. Her interests include distribute file systems, parallel I/O storage systems and graph processing systems. She has more than 50 publications in major journals and international conferences, including FGCS, ACM TACO, SCIENCE CHINA Information Sciences, Chinese Journal of Computers HiPC, ICDCS, HPDC and ICPP.

Hong Jiang received the BE degree from the Huazhong University of Science and Technology, China in 1982, the MASc degree from the University of Toronto, Canada in 1987, and the PhD degree from the Texas A&M University, College Station, in 1991. He is Wendell H. Nedderman Endowed Professor & Chair of Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems and parallel/distributed computing. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACM-TOS, ACM TACO, JPDC, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by NSF, DOD and the State of Nebraska. Dr. Jiang is a Fellow of IEEE.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China in 2001 and 2005, respectively. He is currently a professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing and network storage. He has more than 80 papers to his credit in major journals and international conferences including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), USENIX

ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP and MASCOTS. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, RTSS and IWQoS. He is a senior member of the IEEE and CCF, and a member of ACM and USENIX.

Dan Feng received the BE, ME and PhD degrees in Computer Science and Technology in 1991, 1994 and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE-TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012. She is a member of IEEE and a member of ACM.



Lingling Zhang is currently a PhD student majoring in computer architecture in Huazhong University of Science and Technology, China. Her current research interests include computer architecture, big data and distributed storage systems.

Jun Zhou received the BE degree in computer science and technology from the Huazhong University of Science and Technology (HUST), China in 2011. He is currently working toward the PhD degree in school of computer science and technology in HUST. His interests include big data and distributed storage systems.