The Journal of Supercomputing (2020) 76:549–579
https://doi.org/10.1007/s11227-019-03023-0

# Survey of external memory large-scale graph processing on a multi-core system

**Jianqiang Huang[1,2] · Wei Qin[1] · Xiaoying Wang[2] · Wenguang Chen[1,2]**

Published online: 26 October 2019


**Abstract**
The fast development of big data computing contributes to the fact that large-scale graph processing has become a basic computing model in both academic and industrial communities, and it has been applied in many actual big data computing works, such as social network analysis, Web search, and product promotion. These computing works include large-scale graphs of billions of vertices and trillions of edges. Such scale has brought many challenges to large-scale graph processing. This paper mainly introduces the essential features and challenges of large-scale graph processing and how we can handle billions of edges on a multi-core machine, for which we represent out-of-core processing system and semi-external memory processing systems. This paper also summarizes the key technologies in graph processing systems and forecasts the future development of large-scale graph processing systems.

**Keywords** Graph data processing · Parallel computing · Computing model · Graph algorithms

## 1 Introduction

Graph processing is an abstract expression of the graph structure in the real world on the basis of graph theory and a computing model based on this data structure. Many graph processing problems exist in practical applications, such as shortest path, connected branch, Web page sorting, and friend recommendation. In the early days, the graph processing problem was small in scale, and the main solutions were as follows:

✉ Jianqiang Huang
hjq16@mails.tsinghua.edu.cn

Wenguang Chen
cwg@tsinghua.edu.cn

[1] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

[2] Department of Computer Technology and Application, Qinghai University, Xining 810016, Qinghai, China

- A multi-core machine graph processing algorithm library, such as Stanford GraphBase [1] and BGL [2], was used to limit the scale of graph processing problems.
- Parallel BGL [3] and CGMgraph [4] were used in parallel computing systems. These libraries have made considerable achievements for parallel graph algorithms but are unable to solve the problem of fault tolerance in large-scale distributed systems and cannot guarantee system accuracy and stability.
- To develop a corresponding distributed architecture for a specific graph application to realize graph processing, the code can be optimized for a specific application to achieve high-performance computing. However, when faced with a new graph algorithm or graph representation, a large amount of code needs to be rewritten, which results in low code reuse rate, poor scalability, high requirements for programmers, and poor universality.

In recent years, the increasing scale of graph structure data has been increasing. For example, as of 2013, an average of 5 billion tweets were generated every day on the social platform Twitter [5]. By the second quarter of 2016, Facebook had 1.7 billion monthly active users [6]. Nowadays, more relationships between users are being established, and data usually take up hundreds of gigabytes or even terabytes of storage. Therefore, large graph processing is not only computationally intensive but also memory intensive. Computing large graphs within an acceptable time is difficult.

With the advent of big data, the need for efficient analysis and data value mining for large-scale data has become particularly urgent. In the past decade, an open-source ecosystem with Hadoop [7] as the core and based on the MapReduce [8] programming model has been rapidly popularized with its mature system code implementation and high availability of cluster scale. In the past three years, with the upgrading of the current memory capacity and terabytes of memory becoming the norm, Spark [9] as the core open-source ecosystem increases the performance of mass data processing by order of magnitude based on the RDD [10] memory computing model. However, neither Hadoop nor Spark can adapt to the parallel computing features of data relevance, access locality, and computational iteration in the graph data structure.

In the field of distributed computing, due to its search business requirements, Internet giant Google developed the Pregel [11] graph processing system to meet computational requirements in the context of the graph data structure, which cannot be achieved with high efficiency by the MapReduce programming model. Subsequently, a series of distributed graph processing systems such as GraphLab [12] and PowerGraph [13] came into being. However, the simple multi-core machine system has been attracting more attention than the distributed system, which involves complex programming and management. The multi-core machine system solves the problems of poor disk access performance, low task concurrency, and low device memory bandwidth utilization caused by graph data partition.

In terms of the multi-core machine graph processing system, GraphChi [14], x-stream [15], GridGraph [16], and other systems [17–36] make processing clustered graph data on a multi-core machine possible. In the present multi-core machine graph processing system, due to the increasing graph size and limited

memory capacity of the multi-core machine, the file size of the data to be processed is often several or even dozens of times the size of a multi-core machines memory. Therefore, the system is forced to adopt a data partitioning strategy in data processing. Each time a portion of the data in the diagram is processed, the entire process is completed by performing processing tasks multiple times. The traditional data partitioning strategy involves dividing vertices sequentially from end to end according to the storage order of data files on the disk under the premise that each data block is smaller than the memory capacity and the associated data from the same graph vertex will not be split into two different data blocks.

The most basic traditional partition strategy is to divide the data directly according to the structure of the graph. This strategy can cause two kinds of system problems. One problem is unbalanced computational load, which refers to the condition in which some high-frequency vertices may be distributed together, thereby leading to a large processing load of the entire data block and even to serious I/O overhead. The other problem is low task concurrency, which means that a large number of adjacent vertices will be present in each data block, causing problems such as data access conflicts in parallel computing. As a result, the number of concurrent tasks can be reduced and concurrency greatly reduced.

Compared with BSP models, which have a high synchronization overhead, the asynchronous processing model can avoid unnecessary synchronization overhead and is more suitable for graph data processing. However, the traditional asynchronous processing model is mostly based on locking and graph coloring algorithm to address potential data conflicts in updating [37, 38]. Given the high cost of locking operation in GPU [39–45], graph coloring is the choice to ensure data consistency. Using the traditional coloring algorithm to deal with graphs will produce hundreds of color blocks, among which numerous color blocks contain only a few processing tasks; this condition leads to insufficient use of GPU computing resources.

With the vigorous development of information and communication technology, human society has entered the era of big data. According to Facebook′s assessment, in October 2018, the number of active users per month reached 2.2 billion [46], as shown in Fig. 1, which lists the top seven social network software with monthly active users. A large amount of various types of information can be naturally abstracted into graph structure data [15, 47], thereby allowing corresponding practical problems to be naturally transformed into graph processing problems.

This paper introduces the main features and challenges of current large graph processing. The basic ideas, advantages, and disadvantages of different types of graph processing systems will be compared, analyzed, and summarized from the system perspective. The key techniques of the graph processing system are summarized, and possible research directions for large graph processing systems are given.

## 2 Features and challenges of large graph processing

Graph is a data structure that describes the complex relationship between data. In form, a graph is composed of vertices and edges, which are usually expressed as $G = (V, E)$, where vertex set $V$ represents an object or entity and edge set $E$ represents

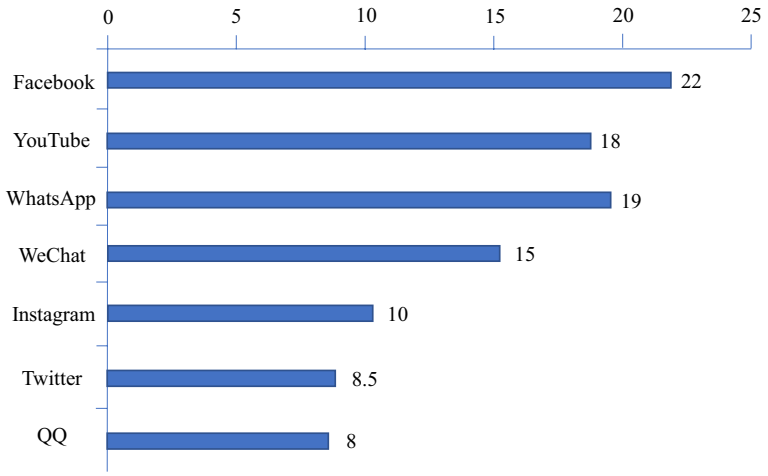



**Fig. 1** Global social network ranking (by number of active users) (unit: 100 million people)

the relationship between objects or entities. Table 1 shows the abbreviations used in this paper. Each vertex and edge may contain additional tags that mark the attribute information of objects, entities, or relationships. Traditional graph theory and graph algorithm focus on the algorithm complexity of the graph problem and the real-scale graph algorithm problem (such as the processing of a multi-core machine with limited memory).

In the current field of big data analysis, the scale of graph data to be processed is often in billions of vertices. Moreover, the graph data structure is complex and changeable, and the graph algorithm is difficult to process efficiently in traditional computing systems. Therefore, a computational model that supports large-scale and efficient graph processing needs to be designed to meet the above challenges. The graph processing model is designed and implemented in accordance with the characteristics of graph data and graph processing, which is commonly used in the graph processing systems. The graph processing model has the following characteristics and challenges, which are different from those of the traditional computing model:

**Table 1** Notations of a abbreviation

| Notation | Meaning |
|---|---|
| $G$ | A graph $G = (V, E)$ |
| $V$ | Vertices in $G$ |
| $E$ | Edges in $G$ |
| $n$ | Number of vertices in $G$, $n = [V]$ |
| $m$ | Number of edges in $G$, $m = [E]$ |
| BFS | Breadth-first search |
| SSSP | Single-source shortest path |
| CSR | Compressed sparse row |
| CSC | Compressed sparse column |
| GPU | Graphics processing unit |
| PSW | Parallel sliding windows |

- Poor locality: The graph represents relationships between different entities that are often irregular and unstructured in practical problems. Therefore, graph processing and memory model do not have good locality. In the current computer architecture, the performance of a program often requires good use of locality. Therefore, how to layout and divide graph data and developing a corresponding computing model to improve data locality are important approaches to improve the performance of graph processing and are also the main challenges.
- Data- and graph structure-driven calculations: Graph processing is entirely driven by the data in the graph. When the graph algorithm is executed, the algorithm is guided by vertices and edges in the graph rather than directly through the code in the program. Therefore, different graph structures will have different computing performances on the same algorithm implementation. Thus, ensuring that different graph structures obtain better processing results on the same system is also a major challenge.
- Unstructured characteristics of graph data: Graph data are often unstructured and irregular in graph processing. When a distributed framework is used for graph processing, the graph needs to be divided and the load must be distributed to each node. This unstructured characteristic of the graph makes achieving effective partitioning of the graph difficult; effective partitioning could attain load balance of storage, communication, and computing. An unreasonable partition causes the unbalanced load between nodes to seriously limit the systems expansibility, and the processing capacity will not be able to meet the computational scale of the system.
- I/O bound: Substantial large graph processing makes storing all the data in the memory impossible. Disk I/O in computation is essential, and most graph algorithms present an iteration. In other words, the whole algorithm needs to be iterated several times, and the whole graph structure needs to be traversed in each iteration, and relatively minimal computing is performed in each iteration. Therefore, a high visiting/computing rate is achieved. In addition, the poor locality of graph processing results in high overhead in waiting for I/O.

The current graph data processing system, whether in a distributed environment or multi-core machine processing system, mainly aims to solve two problems: how to reduce the random access overhead during graph algorithm execution and how to divide graph data efficiently to solve the load imbalance caused by partition. At present, both local research and foreign research focus on the establishment of a graph processing system, data partitioning strategy, and GPU graph data processing [48–51].

### 2.1 Out-of-core graph processing system

With the improved processing and storage capacity of a multi-core computer, as well as extensive research on the graph processing model, some systems for large graph processing on a multi-core computer are proposed. These systems have good graph processing performance and have obvious advantages of low hardware cost and power consumption over distributed systems.

Large-scale multi-core machine graph processing system uses the CPU, memory, and disk on a multi-core machine for large-scale graph processing. Such systems handle large-scale graph data by reducing random disk writes, avoiding high communication overhead, and adopting multi-core and parallel technologies. When the graph size is not very large, the task can be completed within an acceptable time to achieve the time performance of the distributed large-scale graph processing system. Examples of a large-scale multi-core machine graph processing system are GraphChi [14], Grace [17], X-Stream [15], TurboGraph [18], VENUS [19], GridGraph [16], FlashGraph [20], PathGraph [21], etc.

### 2.1.1 GraphChi

GraphChi [14] is a disk-based multi-core machine large graph processing system. In large graph processing, access locality is very poor, which seriously affects the computing performance. In particular, the computational power of the system is limited to multi-core machines. To improve computing performance, GraphChi uses an innovative disk data layout and a corresponding computing model to reduce random access to disks; selective scheduling is used to accelerate the convergence of the algorithm.

GraphChi assumes that the computer memory capacity is limited, that is, (1) the size of the processed graph structure data is much larger than the memory capacity and (2) the computer has enough memory to hold any vertex of graph data and all the adjacent edges of that vertex. Obviously, this assumption is in accordance with the actual situation. The actual large-scale graph data are basically sparse graph, and commonly used sparse graph storage formats, such as row compression (compressed sparse row) or column compression (compressed sparse column), experience the random access problem.

For example, row compression causes the output edge of a vertex to be stored continuously, while column compression makes the input edge of a vertex stored continuously. If only row compression is used, then the output edge of vertices can be accessed continuously and the input edge of vertices needs to be accessed randomly. Likewise, if only column compression is used, then the input edge of vertices can be accessed continuously and the output edge of vertices needs to be accessed randomly. If both row and column compressions are used, then all edges in the graph will be stored in two copies. When the value of the edge is modified, the data synchronization problem occurs. To access the graph data stored in the disk continuously, GraphChi proposed the method of parallel sliding windows (PSW) to solve the random access problem.

GraphChi first preprocesses the graph data before computing, dividing the input graph into multiple shards. Each shard stores all the input edges of the corresponding vertex set and sorts them according to the ID of their source node. When partitioning, the number of edges in each shard should be roughly the same and each shard should be loaded into memory. GraphChi uses a vertex-centric computing model and uses PSW to load data for computing, as shown in Fig. 2. Every interval

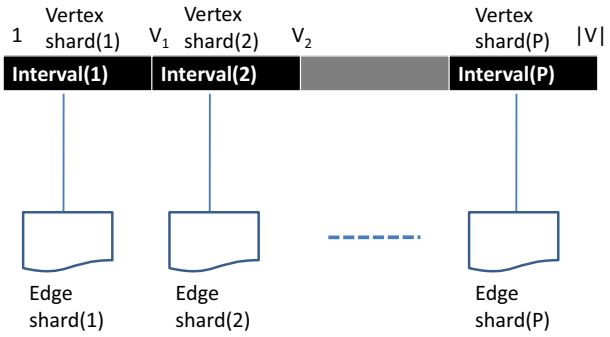



**Fig. 2** Intervals and shards in the graph

computes a subgraph. The input edge (dark gray part) of a point set and its output edge (black rectangle part) in other shards should be read sequentially. This data layout and computing model ensure that I/O is computed each time sequentially. In this way, the values of all vertices in the whole graph are computed in one iteration and iterated many times until the algorithm converges.

In GraphChi, selective scheduling can be used to accelerate the convergence of some vertices in the graph, especially those vertices that change significantly in two adjacent iterations. When the vertex is executing update(), similar to apply() in GraphLab, its neighbor vertices can be added to the scheduler for selective scheduling.

**Algorithm 1** Parallel Sliding Windows (PSW)

```
1: for each iteration do
2:     shards[] = InitializeShards(P)
3:     for interval = 1 to P do
4:         /* Load subgraph for interval, using Alg. 3. Note
5:         that the edge values are stored as pointers to the
6:         loaded file blocks. */
7:         subgraph = LoadSubgraph(interval)
8:         parallel
9:         for each vertex ∈ subgraph.vertex do
10:            /* Execute user − defined update function,
11:            which can modify the values of the edges */
12:            UDF updateVertex(vertex)
13:        end for
14:        /* Update memory − shard to disk */
15:        shards[interval].UpdateFully()
16:        /* Update sliding windows on disk */
17:        for s ∈ 1, .., P, s ≠ interval do
18:            shards[s].UpdateLastWindowToDisk()
19:        end for
20:    end for
21: end for
```

**Algorithm 2** Function LoadSubGraph(p)

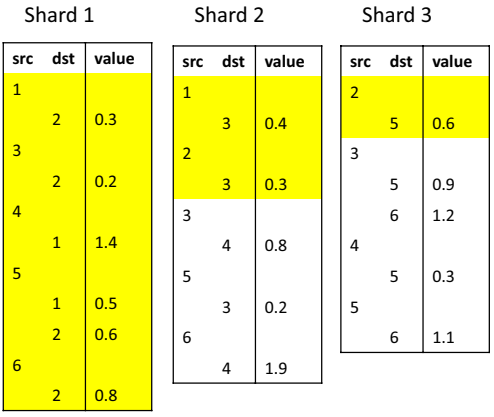
**Require:** *Interval index number p*
**Ensure:** *Subgraph of vertices in the interval p*

```
1: /* Initialization */
2: a = nterval[p].start
3: b = nterval[p].end
4: G = InitializeSubgraph(a, b)
5: /* Load edges in memory − shard. */
6: edgesM = shard[p].readFully()
7: /* Evolving graphs : Add edges from buffers. */
8: edgesM = edgesM ∪ shard[p].edgebuffer[1..P]
9: for each e ∈ edgesM do
10:     /* Note : edge values are stored as pointers. */
11:     G.vertex[edge.dest].addInEdge(e.source, e.val)
12:     if e.source ∈ [a, b] then
13:         G.vertex[edge.source].addOutEdge(e.dest, e.val)
14:     end if
15: end for
16: /* Load out − edges in sliding shards. */
17: for s ∈ 1, .., P, s = p do
18: end for
19: edgesS = shard[s].readNextWindow(a, b)
20: /* Evolving graphs : Add edges from shards buffer p */
21: edgesS ∈ edgesS ∪ shard[s].edgebuffer[p]
22: for each e ∈ edgesS do
23:     G.vertex[e.src].addOutEdge(e.dest, e.val)
24: end for
25: return G
```

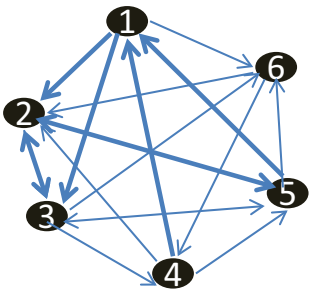
Now we describe a simple example that consists of two execution intervals, as shown in Fig. 3. In this example, we have a graph of six vertices divided into three equal intervals: 1–2, 3–4, and 5–6. Figure 3a shows the initial contents of the three shards. PSW (as shown in Algorithms 1 and 2) starts executing interval 1 and loads the subgraph that contains the edges drawn in bold in Fig. 3c. The first shard places in memory, which is fully loaded. The memory shard contains all the inner edges of vertices 1 and 2, as well as the outer sub-set. Shards 2 and 3 are slide shards, and the window starts with these shards. Shard 2 contains the two outer edges of vertices 1 and 2; shard 3 has only one outer edge. The loaded blocks are shown as shadows in Fig. 3a. After the graph is loaded into the memory, PSW runs the update function of vertices 1 and 2. After the update is performed, the modified block is written to the disk; the updated value is shown in Fig. 3b. Then, PSW moves to the second interval, vertices 3 and 4. Figure 3d shows the corresponding edges in bold, and Fig. 3b shows the loaded blocks in shadows. Now, shard 2 is the memory shard. For shard 3, we can see that the block of the second interval appears after the block loaded in the first interval.

### 2.1.2 X-Stream

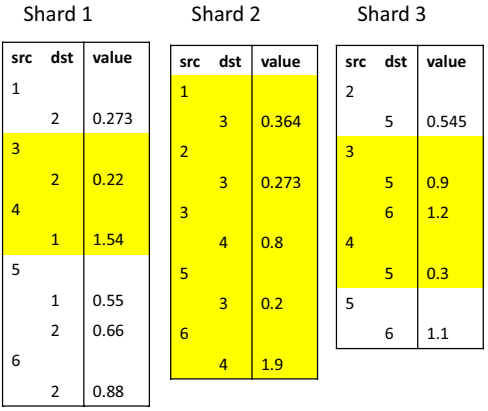
Unlike GraphChi's vertex-centric computing model, X-Stream [15] uses an edge-centric computing model (as shown in Fig. 4), and all states are stored in the point. The computing process of X-Stream is mainly divided into three stages: scatter,



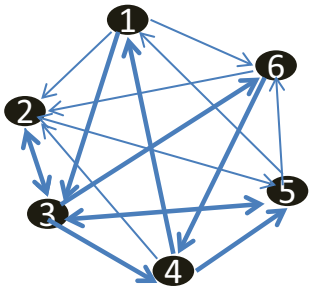

**(a)** Execution interval (vertices 1-2)




**(b)** Execution vertices 1-2




**(c)** Execution interval (vertices 3-4)



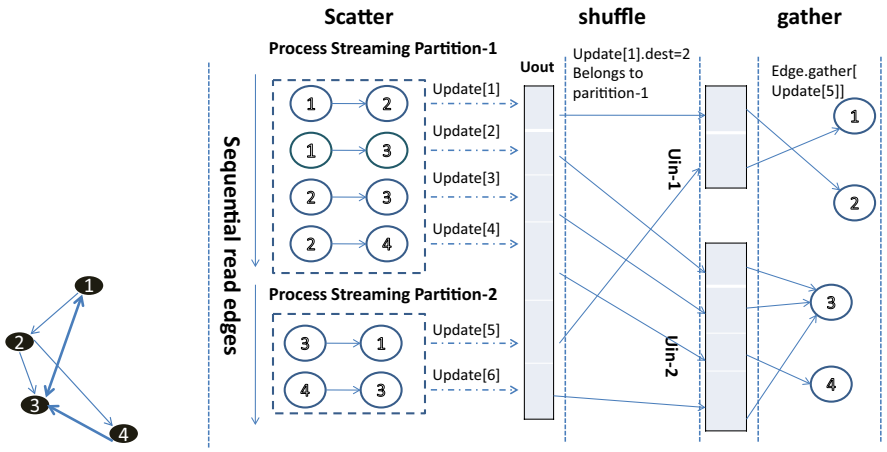

**(d)** Execution vertices 3-4

**Fig. 3** Illustration of the operation of the PSW method on a toy graph (see the text for description)

shuffle, and gather. In scatter phase, X-Stream traverses each edge in order to judge whether the source node of the edge generates update, and if update occurs, the edge is sent out to the destination node through output edge. Shuffle phase refers to the phase of updating data exchange between different partition blocks after graph partition, mainly to reduce the random write overhead in scatter phase.

*Edge-centric computing model*

*scatter phase:*
*for each streaming_partition p*
*read in vertex set of p*
*for each edge e in edge list of p*
*edge_scatter(e): append update to* $U_{\text{out}}$
*shuffle phase:*




**(a)** Graph example **(b)** Graph data processing

**Fig. 4** Graph data processing based on X-Stream

*for each update u in* $U_{\text{out}}$
*let p= partition containing target of u*
*append u to* $U_{\text{in}}(p)$
*destroy* $U_{\text{out}}$
*gather phase:*
*for each streaming_partition p*
*for each update u in* $U_{\text{in}}(p)$
*edge_gather(u)*
*destroy* $U_{\text{in}}(p)$

In gather phase, X-Stream iterates through all the updates generated in scatter phase and updates the state value of the corresponding point. X-Stream uses edge-centric computing model to access edges sequentially, giving full play to the sequential access high-bandwidth of secondary storage media on disk to accelerate graph processing, but the access to the point in X-Stream is random. In order to optimize this and further improve computing performance, X-Stream divides the point set of the graph equally into small sub-sets, and all output edges of each point in sub-sets correspondingly form a partition set of edges. The partition of points is mainly to store all points in each sub-set in memory. In this way, when computing each partition block, the random access overhead of points can be greatly reduced.

After graph partition, each partition block is in scatter phase. First, all update values are written in a local output cache. Then, a shuffle phase begins when all the blocks finish scatter. The main work in shuffle phase is to allocate all the updates of the partition blocks and allocate the updates to the input cache of the corresponding partition blocks as the input of gather phase, and the status of point is updated. Compared with GraphChi, X-Stream makes sequential access to all edges, which

can give full play to the sequential bandwidth speed of secondary storage media on disk, and meanwhile, the preprocessing stage (simple hash graph partition operation) does not require expensive sorting, so it can obtain better graph processing performance.

### 2.1.3 GridGraph

In X-Stream, between scatter and gather phases, a shuffle phase is also required to allocate the updated values generated by each partition in scatter phase to the input cache of the corresponding partition for computing in gather phase. In scatter phase, the updated value will be as large as $O(|E|)$, where $|E|$ represents the number of edges in the graph. Therefore, when the memory runs out, a portion of the cache need to be written to disk first, and the updated value written to disk needs to be re-read into memory in gather phase. Thus, more I/O may be triggered during this process, seriously affecting the performance of the system.

To solve this problem, GridGraph [16] proposes a grid division method as shown in Fig. 5. First, the entire point set is divided into P sub-sets of the same size, and then, the edges are divided into grids with rows and columns, with each row corresponding to all the output edges corresponding to the points in a sub-set and each column corresponding to all the input edges corresponding to the points in a sub-set. Based on this graph partition method, the author puts forward the computing model of double sliding window (as shown in Fig. 5), which is the first iteration of PageRank of graph structure in Fig. 6. To compute the update value of the point, its input source node value needs to be read. To do this, the edges of each grid in column need to be read in sequence from top to bottom for computing. When a column is computed, the point value of a sub-set is computed. Then, the window slides to the next column and continues computing until all grids are traversed.

In this computing model, the updated value computation must conform to the commutative law. In addition, this point value update is local update, which will not produce intermediate update result, greatly reducing I/O. At the same time, the

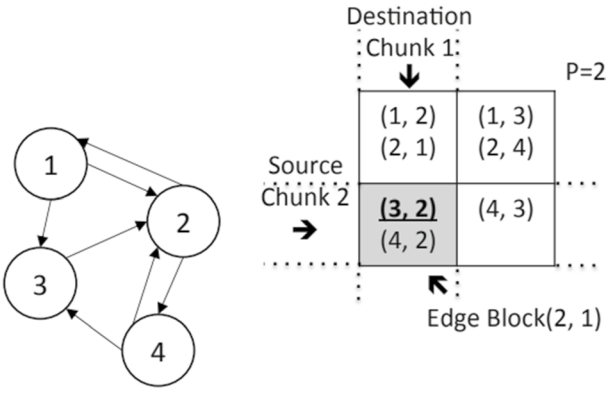



**Fig. 5** Grid representation

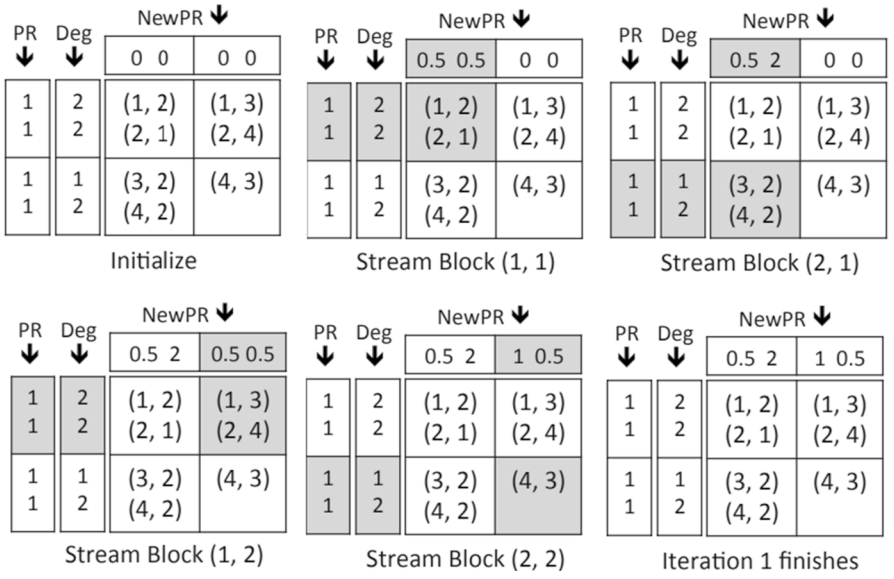



**Fig. 6** Dual sliding windows

locality of point data access will also be improved. The secondary graph partition strategy is used in graph partition. That is, the graph is first divided into Q parts so that the edges of each grid can be stored in memory and then to partition each grid so that each grid can be stored in the last level cache (LLC). In addition, GridGraph supports selective scheduling, which can greatly reduce I/O and improve computing performance in algorithms such as BFS and WCC.

## 2.2 Semi-external memory graph processing system

### 2.2.1 FlashGraph

FlashGraph [20] is a multi-core-machine graph processing system that can process graphs with trillions of nodes on a solid-state disk array. To address the need for larger datasets for graph processing, FlashGraph uses Flash SSD instead of DRAM to hold more data and provide comparable performance. According to the description of this paper, the existing distributed graph processing is either conducted entirely with DRAM, pursuing high performance but high cost, or it is retrograde to use traditional disk as storage but has serious performance impact and the programming model has certain limitations. (Random access to vertex list is not allowed.)

FlashGraph performs in three ways: One is to reduce I/O by making the data it constructs as compact as possible, the other is to adopt sequential I/O as possible, and the third is to parallelize I/O and computing. To do all the three things, a user-space file system is built on top of Linux VFS, primarily to avoid memory copying and to understand I/O. Because issuing I/O to SSD requires a copy of user state and

kernel state, and implementing a simple Page Cache in user state can greatly reduce the memory copy brought by Context Switch. Also, because cache can be application-aware, the cache hit rate can be improved based on the context of the application (Task). In addition, the programming model provided by FlashGraph enables execution below each execution body to the Userspace file system layer, making full use of the Userspace Page Cache for access and avoiding secondary copies of Userspace. Computing intensive applications often have high requirements for memory access, and both locality and space are major optimization points. Therefore, it is necessary to plan the execution space of the whole program into a flat space as much as possible to improve the cache hit rate.

### 2.2.2 VENUS

Although GraphChi can achieve good computing effect in large graph processing, it has the following disadvantages: preprocessing requires sorting the source nodes of the edge, which is costly; the loading and computing of graph data are separate and do not take full advantage of the parallel disk and I/O to improve the computing performance; after sorting edges in shard, the corresponding edges of each point are not adjacent to each other, and the cache locality is not high. Based on the above observations, VENUS [19] proposed vertex-centric flow graph processing model which significantly reduces disk-based graph I/O and greatly improves the situation of system I/O bottleneck. Compared with common big data platform like Hadoop and Spark, VENUS is designed to scale out data and read large data concurrently on large clusters to cope with the large amount of I/O required for large amounts of data. VENUS has a unique technology approach based on extraneous algorithms, known as hierarchical storage-based vertex central fluidized graph processing model, which has broken through the expandability of graph processing system on a multi-core computer and solved the problem of how to effectively utilize disk bandwidth and reduce the access to mass graph data I/O in graph processing. g-shard is similar to the shard in GraphChi, storing all input edges corresponding to a subpoint set. But instead of sorting edges, it stores the same edges at the destination vertex in adjacent locations. V-shard stores values corresponding to all destination and source vertices in a g-shard. In addition, a global table of point values is used from which v-shard reads and writes back the corresponding point values. When the system computes the update value of the point, it does not need to load all the input and output edges into memory at the same time as GraphChi but to load the input edge into memory, and when the node is updated, the updated value does not need to be written into the output edge anymore, which can greatly reduce I/O. In addition, when all the input edges of a point in g-shard are loaded, the value of that point can be computed to overlap I/O. At present, VENUS system analyzes 1 billion orders of magnitude of user log data in Huawei's mobile application market, which supports target customer selection of advertisement and mobile application recommendation service.

### 2.2.3 TurboGraph

Although GraphChi performs much better than disk-based distributed graph engines, it has the following problems:

- Limited parallel processing;
- Step-by-step I/O processing and CPU processing;

The author proposes a new disk-based graph engine, TurboGraph [18], which can efficiently process billion-size graph on a single PC:

- A universal extensible graph engine based on a multi-core computer is proposed, which can make full use of the parallelism between multi-core and I/O and the overlapping processing between CPU and I/O;
- A disk and memory structure for storing a billion-size graph is proposed, which can effectively support graph traversal and bitmap-based operations;
- A fast and extensible core diagram operation is proposed, using the pin-and-slide model;
- Experiments with large datasets show that TurboGraph is four orders of magnitude stronger in performance than GraphChi.

### 2.2.4 PathGraph

PathGraph [21] is a path-centric, multi-core machine graph processing system that improves iterative graph processing for graphs with billions of edges. It models large graphs using collections of tree-based partitions, improving memory and disk access localization for large graph iteration algorithms, and it has developed an optimized compression storage for the parallel computing of the iteration graph. Also, the scatter/gather program model is used to implement the path-centric computing model.

## 3 Key techniques in large graph processing

Current research on graph processing models focuses on performance optimization for a particular pattern or provides the same interface for both patterns to allow users to select according to the characteristics of the algorithm. Reference [6] summarizes the data storage technology and computing model of graph processing and mainly explains the graph processing system and synchronous computing model based on Hadoop platform. Both PowerGraph [13] and Grace [17] systems provide support for both synchronous and asynchronous computing models with the same interface. Different from data parallel computing model represented by MapReduce, the application of graph processing model usually requires the interaction between vertices and several rounds of iteration until all vertex computing converge.

The computation of each vertex is abstracted as three steps of collecting (gather) adjacent vertex data, updating (apply) own data, and activating (scatter) adjacent vertex (GAS). Taking PageRank, which is widely used in Web search and social

impact statistics, as an example, formula (1) shows the algorithm of Web page $v$ to compute its ranking value. $d$ is an attenuation coefficient, usually at 0.85; $n$ is the number of pages; $v.in\_ngbrs$ is all pages that point to $v$; $u.noutngbrs$ is the number of pages that the Web page $u$ points to.

$$v.rank = \frac{1-d}{n} + d \times \sum_{u \in v.in\_ngbrs} \frac{u.rank}{u.noutngbrs} \tag{1}$$

Based on formula (1), Algorithm 3 gives the realization of PageRank under GAS computing model. Each Web page is regarded as a vertex in the graph. Users need to maintain a ranking value rank for each vertex and provide three function interfaces of gather, apply, and scatter. The gather ($v$, $u$) function is used to collect the ranking value of the adjacent vertex $u$ of vertex $v$, the apply ($v$) function uses the sum of the neighboring vertex value to update the ranking value of the vertex $v$, and the scatter ($v$, $u$) function determines whether to activate the adjacent vertex $u$ according to the change in the ranking value of vertex $v$. During the computing of vertex $v$, the graph processing framework executes the gather function (lines 1–3) for all adjacent vertices pointing to vertex $v$ to collect the ranking values; the apply function is then called on vertex $v$ to update the ranking value (lines 5–9); finally, the scatter function (lines 12) is called for all vertices pointed by vertex $v$ to activate adjacent vertices.

**Algorithm 3** :Pseudo-code of the vertex update function for weighted PageRank.

**Input:** All intervals vertex-shards and edge-shards of graph G, optional initialization data.
**Output:** Desired output results.

```
1: function UPDATE(vertex)
2:     Initialize(vertex-shards);
3:     repeat
4:         v[i] =  read values of out-edges of vertex i ;
5:         vertex.value = f(v[i]) ;
6:        if Δf(v[i]) ≠ 0 then  f(v[i])  = Δf(v[i]) + f(v[i]) ;
7:            for each edge of vertex do
8:                edge.value = f(vertex.value, edge.value));
9:               Δf(v[i]) = 0 ;
10:           end for
11:       end if
12:       PassingMessage(vertex) ;
13:       remove outgoing edges of i
```

Accumulate $\Delta V_i$ to vertex $i$ and perform an update operation to use the update of $\Delta V_j$ of the neighbor vertex, $j$, followed by resetting the change of information in vertex $i$. When the operation on vertex $i$ is completed, the edge data of vertex $i$ are deleted from memory to free memory space for other uncomputed vertex edge data. This activity is repeated until the algorithm converges.

### 3.1 Computing model

With the increasing scale of graph data, the processing and storage capacity of a multi-core server cannot meet computing needs; thus, the distributed platform becomes the inevitable choice. The input graph needs to be evenly distributed

across all the machines in the cluster, and distributed graph processing is completed through messaging. The messaging of graph processing can be summarized in two ways: direct messaging between adjacent vertices and messaging between vertices and read-only replication. Direct messaging, while simple, does not support the active capture of the vertex to contiguous vertex data (pull pattern) and can send updates in one direction only. Therefore, dynamic computing and incremental computing cannot be implemented, and all vertices must participate in the whole computing process even if most of them have converged.

The system based on vertex backup simulates distributed shared memory by creating local read-only replication for cross-machine vertices to meet the needs of adjacent vertex data access during computing, and the data of vertex and backup are kept consistent through messaging. The converged vertex does not need to participate in computing, and redundant computing is avoided. Currently, a large number of distributed graph processing models (e.g., GraphLab [12], PowerGraph [13], and GraphX [52]) adopt the message delivery model based on vertex backup. The same message delivery model based on vertex backup is used in this article.

The engine of the existing distributed graph processing framework can be divided into synchronous and asynchronous according to different control flow and data flow patterns. The control flow pattern determines how vertex computing is scheduled, while the data flow pattern determines when adjacent vertex data values are used. The pros and cons of both synchronous and asynchronous computing patterns are discussed as follows.

#### 3.1.1 Synchronous computing model

Synchronous computing model [53] (briefly, synchronous model) adopts synchronous control and data flow, dividing the whole computing process into several rounds of iteration, and all living vertices in each round are computed using the adjacent vertex value in the previous round. Global synchronization between rounds ensures that vertex computing of the previous round and vertex value update (in which the vertex synchronizes the data to the backup vertex via messaging) has been performed on all machines. In Algorithm 4, $V$ is the set of vertices to be computed for this vertex, $V'$ is the set of vertices to be excited by this vertex, $M$ is the message set generated by this computing, and $A_v$ is the set of contiguous vertices to be activated by vertex $V$. Compute() calls the user-implemented graph algorithm code to complete computing on vertex $v$, updates vertex values, and produces message m and the set of vertices that needs to be activated, $A_v$(line 5).

**Algorithm 4** Synchronous Computing Model

```
1: while iteration <= max_iteration do
2:     if V! = ∅ then V' = ∅
3:     end if
4:     for each v ∈ V do
5:         (v, m, A_v) = v.compute()
6:         M = M ⋃ m
7:         V' = V' ⋃ A_v
8:     end for
9:     barrier()
10:    V = V'
11:    iteration++
```

As shown in Algorithm 4, all activated vertices in each round are computed using the values of adjacent vertices up to the previous round, and all messages are sent and received in batches (line 9) through the exchange(). The vertices activated by each vertex computing will be marked for the next round of computing (line 7). New values for all vertices are visible simultaneously after global synchronization. The computing ends when no vertex is activated for the next round of computing after the previous round ends. Given that each round in the synchronization model only has to traverse a living vertex and record an activated vertex, and access to vertex data during computing is not protected (the vertex value in the previous round is always used), the scheduling overhead is small. However, each round of computing can use only the value in the previous round, which results in slow convergence of the whole computing process and additional vertex computing times and computing rounds. As the number of convergent vertices increases, the proportion of effective computing time decreases rapidly, while the global synchronization overhead takes up most of the execution time in each round causing serious performance loss.

#### 3.1.2 Asynchronous computing model

The asynchronous computing model (referred to as asynchronous model) employs asynchronous control and data flow to maintain distributed scheduling queues for all active vertices [54]. As shown in Algorithm 5, the asynchronous model takes a vertex $V$ from the scheduling queue $V$ each time for computing (line 2). The set of vertices $A_v$ activated by vertex $v$ is returned to the scheduling queue (line 4). The values updated by vertex $v$ can be used immediately by other local vertices and used by remote vertices (lines 6 and 7) through message exchange within a certain interval without waiting for global synchronization. The computing ends when the scheduling queue is empty. Vertex computing in the asynchronous model can use a relatively new adjacency vertex value; thus, the computing process converges faster with few iterations of vertex computing. However, the performance loss is serious due to the need to maintain consistency of scheduling queues and data access in a distributed environment.

**Algorithm 5** Asynchronous Computing Model

```
1: while V! = ∅ do
2:     v = dequeue(V)
3:     (v, m, A_v) = v.compute()
4:     enqueue(V, A_v)
5:     M = M ⋃ m
6:     if size(M) > interval then
7:         exchange(M);
8:     end if
```

### 3.1.3 Hybrid computing model

On the basis of the analysis and comparison of synchronous and asynchronous computing models, this paper proposes the hybrid computing model [55] (referred to as hybrid model), that is, asynchronous data flow (messaging) is used on the basis of synchronous control flow (task scheduling). Combining the advantages of the two existing models, this model can achieve relatively fast convergence speed while reducing scheduling overhead, thus improving the performance of the distributed graph processing system. As shown in Algorithm 6, this hybrid computing model adopts the vertex scheduling similar to the synchronous model. Several iterations of computing are still performed until the vertices are not activated, and each round passes through all the living vertices in turn for computing. Unlike the synchronous model, however, the hybrid computing model uses asynchronous messaging to enable the vertex to be computed by using a new contiguous vertex value. Each newly computed vertex value can be immediately available to the local vertex and used by the remote vertex (lines 8 and 9) at some interval through message exchange. This hybrid model does not need to maintain the distributed scheduling queue in parallel, thereby avoiding the scheduling overhead of the asynchronous model. Given that vertex computing can partially use the new values derived from adjacent vertices in this round of computing, it can converge faster than the synchronous model, thereby reducing the number of vertex computing and iteration rounds. The hybrid model, which combines the advantages of synchronous and asynchronous models and overcomes their disadvantages, is an ideal model for graph processing engines in a distributed environment.

**Algorithm 6** Hybrid Computing Model

```
1: while iteration <= max_iteration do
2:     if V! = ∅ then V' = ∅
3:     end if
4:     for each v ∈ V do
5:         (v, m, A_v) = v.compute()
6:         M = M ⋃ m
7:         V' = V' ⋃ A_v
8:     end for
9:     if size(M) > interval then exchange()
10:    end if
11:    barrier()
12:    V = V'
13:    iteration++
```

### 3.2 Communication model

In the communication model of message passing [56–58], the state of the point in the algorithm is saved locally, and the state of the point on other machines is updated via message passing. The communication model of message passing is used in Pregel and Giraph. To ensure that all updated data are available, a synchronization operation needs to be added between two iterations. In the communication model of shared memory, each processing unit allows concurrent access and modification of data at the same address. In some distributed computing systems, such as GraphLab and PowerGraph, virtual shared memory is used to achieve transparent synchronization between computing nodes. Virtual shared memory is implemented in these graph processing systems by using ghost vertex. In the implementation strategy for ghost vertices, each point in the graph has a working node of its own, and other working nodes have copies of that point. Therefore, in this communication model, data consistency needs to be considered when multiple working nodes access the same memory address concurrently.

### 3.3 Graph partition

Graph partition [14, 16, 59, 60] is a key problem in efficient graph processing. Generally, an ideal graph partition situation is that work nodes have the same amount of work and the communication between work nodes is minimal, but this is a NP-hard problem. Currently, common graph partition algorithms are divided into three categories.

- In the first category, the input graph data are first preprocessed to transform the initial graph data into a specific storage format, which improves the visiting locality of graph processing or lessens the amount of space occupied by the graph data.
- In the second category, dynamic repartition is performed during algorithm execution. Given that the algorithms behavior before execution cannot be predicted, this dynamic partition strategy can be adjusted according to the execution state of the existing algorithm to improve the performance of the system. This dynamic partition strategy needs to divide the graph several times, thereby introducing graph partition overhead.
- In the third category, edge-cut partition and vertex-cut partition are used. Edge-cut partition evenly divides the points in the graph and ensures a minimum number of edges across different partition blocks. Vertex-cut partition evenly divides the edges and keeps a minimum number of points across different blocks. Many real-life large graphs conform to the power law distribution. Therefore, unlike edge-cut partition, the vertex-cut partition can ensure the load balance of the system, but the graph processing system needs to use an edge-centered computing model such as PowerGraph.

### 3.4 Load balance

The load balance algorithm can be divided into static load balance and dynamic load balance [36, 61–65]. The static load balance is used to allocate tasks before the implementation of the algorithm. However, because the algorithm cannot predict its specific behavior before execution, load imbalance may occur during the implementation of the algorithm. The dynamic load balance is improved based on the static load balance, that is, in the running process of the algorithm, the working nodes with fewer tasks in the system can steal tasks from the working nodes with large tasks to achieve load balance and further improve the performance of the system.

### 3.5 Fault tolerance

Fault tolerance is a problem to be solved in distributed graph processing systems [65–69]. In the distributed processing system, each machine will have a certain probability of error failure, and if it is not handled, then it will have a serious impact on the system. The common distributed graph processing system uses the master–slave node approach, in which the master node is responsible for the management and scheduling of the whole system and the slave node is responsible for a specific computing task. The main fault-tolerant strategies include multiple copy strategy and log redo strategy. In the multiple copy strategy, when the primary work node executes its task, another working node as a replica will perform the same task; when the primary work node fails, the replica takes over the tasks of the primary work node. This fault tolerance method has almost no error recovery time but consumes a large amount of computing and memory resources. In the log redo strategy, checkpoint or log is used to record the computing operation of work nodes. When the machine fails, the recorded operation can be repeated for recovery. This recovery method will consume a certain amount of recovery time, but the consumption of computing and memory resources is relatively small.

## 4 Performance evaluation comparison

We evaluate large-scale graph processing systems on several real-world social graphs and network graphs, which requires to analyze a large number of graph algorithms to discover their common features. First four commonly used large-scale figure structure data processing algorithms are introduced here: Web page rank (PR) algorithm, breadth-first search (BFS) algorithm, multi-core source shortest path (SSSP) algorithm, and connected components (CC), and important operators of each algorithm are listed in table form.

The PR algorithm is a link analysis algorithm in which each vertex in the graph corresponds to a numerical weight, and the importance of each vertex in the whole vertex set is measured by comparing their weight values. Before the algorithm starts execution, the weight value of the vertex is initialized to 1. The algorithm iterates to execute the given number of iterations or the weight value of all vertices is no longer

changed as the termination condition. Formula (1) shows how the weight value of each vertex is computed, where $v.rank$ represents the weight value of vertex $v$, $a$ as a constant, and $u.OutDeg$ as the output value of point $u$.

BFS is a widely used graph data search algorithm and is also the basic algorithm of the Graph500 test set. In this algorithm, each vertex corresponds to a distance value. Before the algorithm starts execution, the distance value of the given starting vertex is initialized to 0, and the distance value of the remaining vertex is initialized to infinity (*INF*). The algorithm executes iteratively, with the distance value of all vertices no longer changing as the termination condition. Formula (2) shows how to compute the distance value of vertex $v$ to $v.dist$. .

$$v.dist = \sum_{u|(u,v)\in E \wedge u.dist \neq INF} \min(v.dist, u.dist + 1) \tag{2}$$

SSSP is a commonly used method of traversing graph data. In this algorithm, each edge has a weight value, and each vertex corresponds to a distance value dist. Given a starting vertex, the shortest distance from other vertices to that starting vertex is computed. Before the algorithm starts execution, the distance value of the given starting vertex is initialized to 0, and the distance value of the remaining vertex is initialized to infinity (*INF*). The algorithm executes iteratively, with the distance value of all vertices no longer changing as the termination condition. Formula (3) shows how to compute the distance value of vertex $v$ to $v.dist$.

$$v.dist = \sum_{u|(u,v)\in E \wedge u.dist \neq INF} \min(v.dist, u.dist + e.weight) \tag{3}$$

CC is a commonly used graph structure data analysis algorithm. In this algorithm, each vertex corresponds to a label value, and the label value of vertex is initialized to the identification number of this vertex. The algorithm executes iteratively, with label values of all vertices no longer changing as the termination condition. The following formula shows how to compute the label value of vertex $v$ to $v.label$.

$$v.lable = \sum_{u|(u,v)\in E} \min(v.lable, u.lable) \tag{4}$$

### 4.1 Test environment

This computer is equipped with an Intel i7-4790k (4.0 GHz) processor, and the CPU contains four cores (with two threads running on each core) and 8 MB L3 caches. The computer is equipped with 16 GB physical memory, four 1 TB SATA-III disks that are organized into a RAID-0 disk array, and an 800 GB SSD that runs on Ubuntu 12.04 system.

### 4.2 Test dataset and test graph algorithm

For each system, we run BFS [70], WCC [71], SpMV [72], and PR on four datasets: LiveJournal [73], Twitter [74], UK [75], and Yahoo [76], as shown in Table 2.

All graphs are real-world graphs with power law distributions. Live Journal and Twitter are social graphs that show relationships between users in each social network. UK and Yahoo! are network diagrams that consist of hyperlinks between Web pages with a larger diameter than a social graph. We run BFS and WCC until they converge, that is, no more vertices can be found or updated. For SpMV, we run an iteration to compute the multiplication result. For PR, we run 20 iterations on each graph.

### 4.3 CPU utilization analysis

Figures 7, 8, and 9 show the comparison of CPU utilization when different multi-core machine graph calculation systems run different graph algorithms to process graph data. In graph processing, multi-core machine graph processing engine is

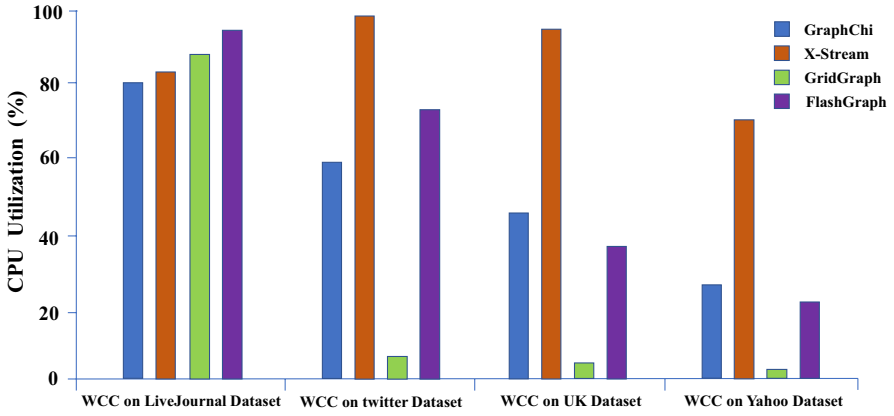



**Fig. 7** GraphChi, X-Stream, GridGraph, FlashGraph average CPU utilization for processing PageRank

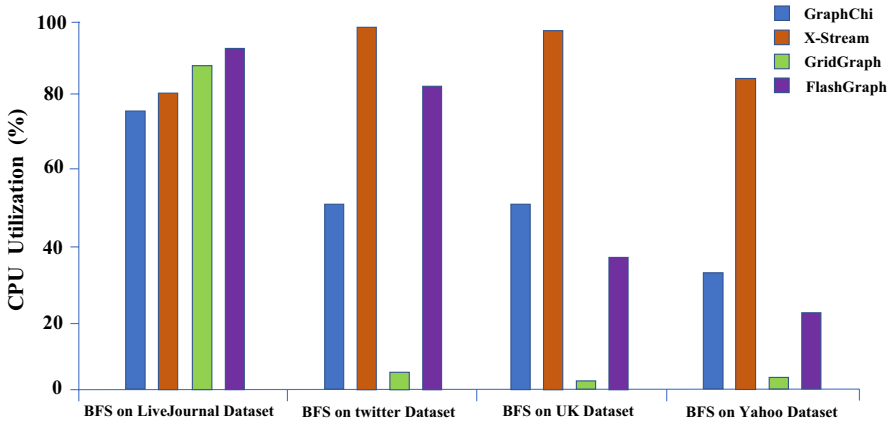



**Fig. 8** GraphChi, X-Stream, GridGraph, FlashGraph average CPU utilization for processing BFS

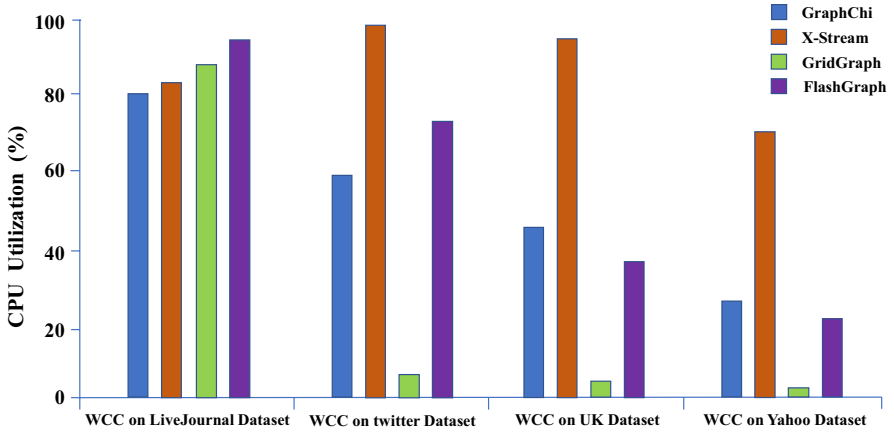



**Fig. 9** GraphChi, X-Stream, GridGraph, FlashGraph average CPU utilization for processing WCC

generally divided into the in-memory computing engine and out-of-core engine. If the pending graph data can be stored in the memory, completely so the graph processing system is based on the memory; otherwise, each part will be from the peripheral storage data into the memory . The LiveJournal dataset shows the average CPU utilization of each graph processing system of the in-memory computing engine. In this case, each individual graph processing system has high CPU utilization. The Twitter, UK, and Yahoo! datasets show a comparison of CPU utilization in the out-of-core engine of individual graph processing systems. Among them, FlashGraph [20] is a semi-external memory graph engine that stores the vertex state in the memory and edge lists on SSDs. As can be seen from these figures, GridGraph [16] has the lowest CPU. For X-Stream, the system adopts an asynchronous direct I/O approach and bypasses the page cache of OS. X-Stream creates a thread that specifically tasks I/O with accessing the disk, and it overlaps the space between GridGraph system performances, although GridGraph is superior to X-Stream with in-place because GridGraph directly updates the status of vertices, whereas X-Stream [15] does so indirectly. I/O updates the update strategy and needs to update the value of the source vertex storage to update the tables (update list). Then, it needs to visit the update table to collect and update the value of the source vertex status to the destination utilization.

### 4.4 Data locality analysis

Figures 10, 11, and 12 show the comparison of data locality of these datasets in different multi-core machine graph processing systems by using the above graph algorithm. As can be seen from these figures, GridGraph has the best data locality performance, which is one of the reasons GridGraph performs better than GraphChi and X-Stream in processing Twitter and UK with BFS and WCC.

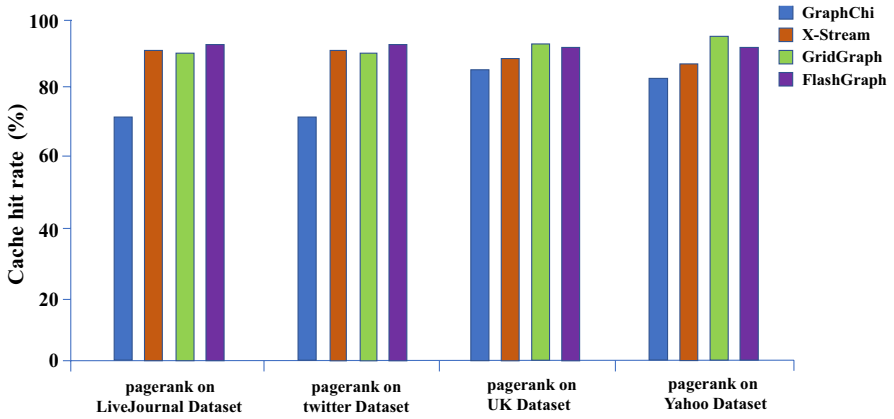



**Fig. 10** GraphChi, X-Stream, GridGraph, FlashGraph average cache hit rate for processing PageRank

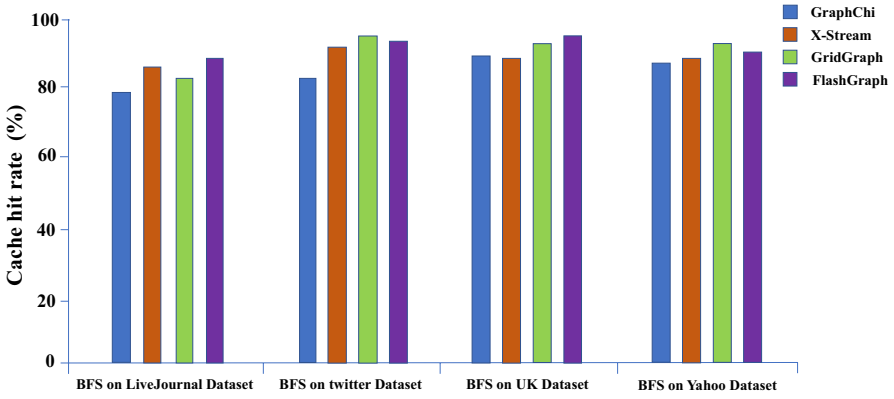



**Fig. 11** GraphChi, X-Stream, GridGraph, FlashGraph average cache hit rate for processing BFS

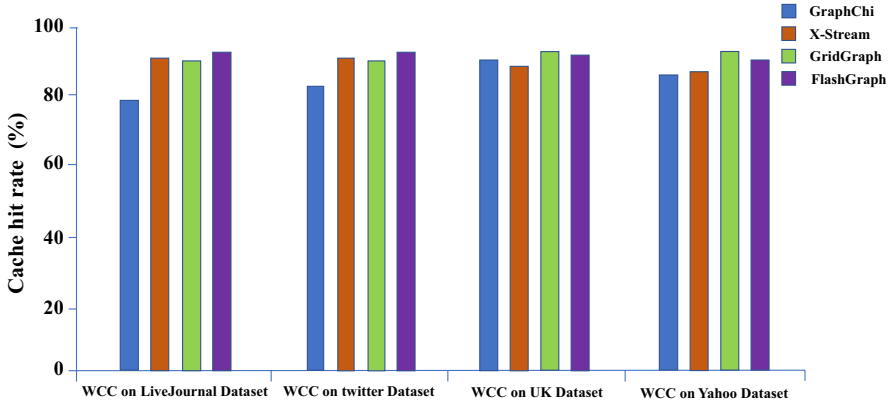



**Fig. 12** GraphChi, X-Stream, GridGraph, FlashGraph average cache hit rate for processing WCC

### 4.5 I/O analysis

We compared four systems: GraphChi, X-Stream, GridGraph, and FlashGraph. As shown in Fig. 13, FlashGraph performs better on different datasets than GraphChi, X-Stream, and GridGraph. To better understand this result, we analyzed the total amount of I/O of BFS, WCC, and PR algorithms in different graphs, as shown in Fig. 13. Specifically, compared with GraphChi, S-Stream, and GridGraph, the I/O data volume of Twitter, UK 2007, and Yahoo! is reduced by 10 instantly, and the vertex state does not need to be written back to the disk.

### 4.6 Comparison of large-scale multi-core machine graph processing systems

We evaluate the processing performance through a comparison with the latest version of GraphChi, X-Stream, GridGraph, and FlashGraph. Table 3 presents the performance of the chosen algorithms on different graphs and systems with memory limited to 8 GB to illustrate applicability. Under this configuration, only the LiveJournal graph can fit the memory, whereas the other graphs require access to disks. We can see that GridGraph outperforms GraphChi and X-Stream.

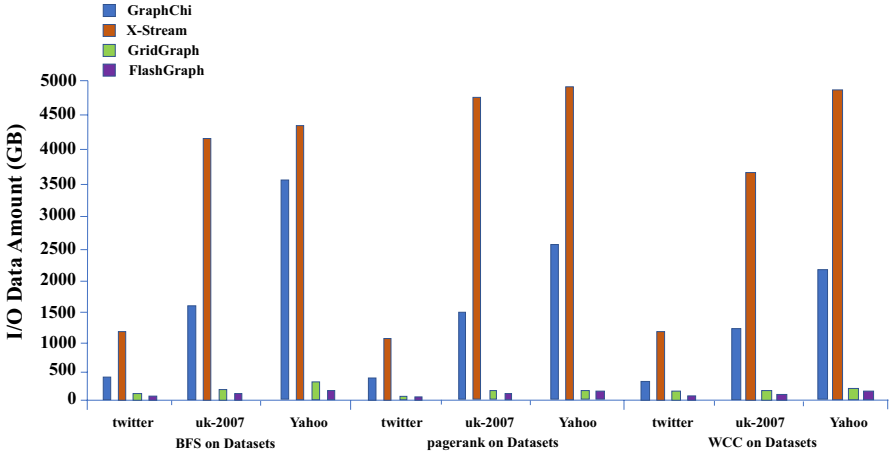



**Fig. 13** GraphChi, X-Stream, GridGraph, FlashGraph I/O data Amount for processing different graph algorithm

**Table 2** Graph datasets used in evaluation

| Dataset | *V* | *E* | Data size |
|---|---|---|---|
| LiveJournal | 4.85M | 69M | 527 MB |
| Twitter | 41.6M | 1.47B | 11 GB |
| UK | 106M | 3.47B | 28 GB |
| Yahoo | 1.41B | 6.64B | 50 GB |

**Table 3** Execution time (in seconds) with 8GB memory

| | BFS | WCC | SpMV | PageRank |
|---|---|---|---|---|
| *LiveJournal* | | | | |
| GraphChi | 21.96 | 46.77 | 11.03 | 52.88 |
| X-Stream | 6.61 | 15.05 | 6.48 | 18.16 |
| GridGraph | 2.69 | 4.16 | 2.09 | 12.46 |
| FlashGraph | 2,88 | 4.29 | 1.82 | 10.89 |
| *Twitter* | | | | |
| GraphChi | 435.2 | 466.9 | 271.8 | 1261 |
| X-Stream | 433.9 | 1203 | 141.6 | 1784 |
| GridGraph | 201.6 | 288.2 | 48.98 | 534.7 |
| FlashGraph | 100.6 | 141.2 | 32.1 | 196.8 |
| *UK* | | | | |
| GraphChi | 2773 | 1782 | 413.9 | 2081 |
| X-Stream | 8086 | 12049 | 386.2 | 43,768 |
| GridGraph | 1839 | 1702 | 112.2 | 1341 |
| FlashGraph | 296 | 1126 | 99 | 968 |
| *Yahoo* | | | | |
| GraphChi | – | 114185 | 2669 | 13,066 |
| X-Stream | – | – | 1071 | 9948 |
| GridGraph | 16,786 | 3598 | 258.4 | 4702 |
| FlashGraph | – | – | – | – |

'–' Indicates that the corresponding system failed to finish execution in 48 h

## 5 Conclusions and opportunities

The wide application of large-scale graph data processing promotes the rapid development of large-scale graph processing systems. In the era of big data, an increasing number of problems will need to be solved by using large-scale graphs. Therefore, researchers in industry and academia have developed numerous large-scale graph processing systems. Multi-core systems are characterized by simple programming and computing models, low hardware overhead, and limited computing power that cannot meet some computing requirements. From the perspective of computing models, two main types of computing models for large graph processing exist: point-centered and edge-centered computing models. GraphChi mainly USES vertex-centric computing models that are easier to program and understand. Edge-centered computing models such as X-Stream are also used. In addition to these two main computing models, other systems proposed new computing models to improve system performance on the basis of data locality. However, in essence, these computing models are based on vertex-centric computing models that are modified for the layout of data. This paper summarizes the research on the existing large-scale graph processing system. Although large-scale graph processing systems have made some achievements, studies can be conducted further from the following aspects:

- A large-scale distributed graph processing system with hybrid computing model. In recent years, some MapReduce large-scale distributed graph processing systems based on the MapReduce model and large-scale distributed graph processing systems based on BSP and GAS models have been developed, all of which have their own advantages and defects. How to integrate the advantages of each model and develop a large-scale distributed graph processing system with better overall performance and high efficiency, expansibility, reliability, and flexibility is a research problem that is worthy of further discussion;
- Partial synchronous or restricted asynchronous large-scale distributed graph processing system. Synchronous paradigm scheduling is simple and correctness is guaranteed, while asynchronous paradigm computing converges quickly and has fast execution speed. Partial synchrony or restricted asynchrony has the advantages of synchronous and asynchronous paradigms. Designing a large-scale distributed graph processing system with partial or restricted asynchrony faces a considerable challenge given that the design needs to balance the execution speed and accuracy of graph processing to ensure that it has the accuracy of the synchronous paradigm and the quick execution of the asynchronous paradigm;
- Large-scale distributed graph processing system integrating the traditional parallel computing framework. The existing large-scale graph processing system is mostly coarse-grained parallel. A good research topic is the identification of the advantages of traditional parallel computing frameworks such as MPI, SMP, MMPS, OpenMP, GPU, CUDA, OpenCL, OpenMP, and OpenACC and their integration into large-scale graph processing systems for fine-grained parallel optimization and to improve existing large-scale graph processing systems;
- Real-time incremental big data graph processing system. Most existing large-scale graph processing systems are global batch processing systems that are oriented to a static graph structure. Thus, these systems have difficulty meeting the real-time requirements of dynamic graph structures. They give priority to the efficiency of offline processing or the optimization of graph processing and pay little attention to how to deal with big data graph processing in real time incrementally. They are also inefficient in frequently updating large-scale real-time graph data and graph structure. To meet the requirements of applications such as Twitter, Facebook, and Sina Weibo and of real-time graph processing of dynamic changes in graph structure, a real-time incremental big data graph processing system needs to be developed;
- Big data graph processing system based on parameter server architecture. Under the parameter server architecture, new copy consistency of parameters should be designed to balance the correctness and concurrency of the algorithm, that is, to improve the concurrency of the whole system as much as possible on the premise of ensuring the correctness of the algorithm and further design the big data graph processing system based on the parameter server architecture;
- Large datasets of large-scale graphs. At present, although some graph datasets can be used to evaluate large-scale graph processing systems, they are not large in scale. In the era of big data, multiple big datasets of various types of large-scale graph processing systems need to be developed to test on these big datasets and for further performance comparison;

- Analysis and evaluation of large-scale graph processing system. For the existing large-scale graph processing system, the present theoretical analysis from different computing models, key technologies, and methods is not sufficiently comprehensive. A comparison of the experimental evaluation of the existing large-scale graph processing system on the existing graph dataset is still lacking. The existing large-scale graph processing system and graph dataset can be integrated to develop the corresponding evaluation model, and comparative analysis and quantitative comparison can be conducted from theoretical analysis and experimental evaluation. In the aspect of multi-core machine graph processing system, due to limited computing power, a hot research topic is the use of an effective graph partition strategy and a computing model that matches this strategy to enhance the localization of computing. The multi-core characteristics of the machine should be fully used in parallel I/O and computing and to improve the parallelism in computing, which is a topic that also deserves further research.

In conclusion, in the era of big data, with the complexity of graph data scale, representation, and organization form of graph data and graph processing tasks, new big data graph processing systems are believed to be constantly generated according to application requirements. Although many research achievements have been made in the field of the large-scale graph processing system, room for innovation and optimization still exists in the area of graph processing platforms, graph algorithms, and graph datasets. This paper aims to help researchers understand the field and provide guidance for the improvement of large-scale graph processing systems.

**Acknowledgements** The authors are grateful to the reviewers for valuable comments that have greatly improved the paper. This paper is partially supported by the Open Project of State Key Laboratory of Plateau Ecology and Agriculture, Qinghai University (No. 2020-ZZ-03), the "Qinghai Province High-end Innovative Thousand Talents Program Leading Talents" and the National Natural Science Foundation of China (Nos. 61762074 and 61962051), and National Natural Science Foundation of Qinghai Province (No. 2019-ZJ-7034).

## References

1. Knuth DE (1993) The Stanford GraphBase: a platform for combinatorial computing. ACM Press, New York
2. Siek JG, Lee L-Q, Lumsdaine A (2002) The boost graph library: user guide and reference manual. Addison-Wesley, Boston
3. Gregor D (2005) Lumsdaine A The parallel BGL: a generic library for distributed graph computations. Proc Parallel Object Oriented Sci Comput 2:1–18
4. Chan A, Dehne F, Taylor R (2005) CGMGRAPH/CGMLIB: implementing and testing CGM graph algorithms on PC clusters and shared memory machines. Int J High Perform Comput Appl 19(1):81–97
5. Twitter Usage Statistics. http://www.internetlivestats.com/twitter-statistics/. Accessed October (2016)
6. Monthly Active Facebook Users. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. Accessed October (2016)
7. Shvachko K, Kuang H, Radia S et al (2010) The hadoop distributed file system. In: Proceedings of the 26th IEEE symposium on mass storage systems and technologies, pp 1–10

8. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
9. Zaharia M, Chowdhury M, Franklin MJ et al (2010) Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, p 10
10. Zaharia M, Chowdhury M, Das T et al (2012) Resilient distributed datasets: a fault tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p 2
11. Malewicz G, Austern MH, Bik AJ et al (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM New York, NY, USA, pp 135–146
12. Low Y, Bickson D, Gonzalez J et al (2012) Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc VLDB Endow 5(8):716–727
13. Gonzalez JE, Low Y, Gu H et al (2012) PowerGraph: distributed graph parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, pp 17–30
14. Kyrola A, Blelloch GE, Guestrin C (2012) GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, pp 31–46
15. Roy A, Mihailovic I, Zwaenepoel W (2013) X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM New York, NY, USA, pp 472–488
16. Zhu X, Han W, Chen W (2015) GridGraph: largescale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of the 2015 USENIX Annual Technical Conference, pp 375–386
17. Prabhakaran V, Wu M, Weng X et al (2012) Managing large graphs on multicores with graph awareness. In: Proceedings of the 2012 USENIX Annual Technical Conference, pp 41–52
18. Han WS, Lee S, Park K et al (2013) TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, pp 77–85
19. Cheng J, Liu Q, Li Z et al (2015) VENUS: vertex-centric streamlined graph computation on a single PC. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering IEEE, pp 1131–1142
20. Zheng D, Mhembere D, Burns R et al (2015) FlashGraph: processing billion-node graphs on an array of commodity SSDs. In Proceedings of the 13th USENIX Conference on File and Storage Technologies, pp 45–58
21. Yuan P, Xie C, Liu L et al (2016) PathGraph: a path centric graph processing system. IEEE Trans Parallel Distrib Syst. https://doi.org/10.1109/TPDS.2016.2518664
22. Feng Z, Heng L, Jidong Z, Jie C, Dingyi X, Jizhong L, Yunpeng C, Xiaoyong D (2018) An adaptive breadth-first search algorithm on integrated architectures. J Supercomput 74(11):6135–6155
23. Zhang M, Wu Y, Zhuo Y, Qian X, Huan C, Chen K (2018) Wonderland: a novel abstraction-based out-of-core graph processing system. In: ASPLOS, pp 608–621. ACM
24. Vora K, Xu GH, Gupta R (2016) Load the edges you need: a generic I/O optimization for disk-based graph processing. In: USENIX ATC, pp 507–522
25. Vora K, Gupta R, Xu G (2017) KickStarter: fast and accurate computations on streaming graphs via trimmed approximations. In: ASPLOS, pp 237–251
26. Maass S, Min C, Kashyap S, Kang W, Kumar M, Kim T (2017) Mosaic: processing a trillion-edge graph on a single machine. In: EuroSys, pp 527–543. ACM
27. Ai Z, Zhang M, Wu Y, Qian X, Chen K, Zheng W (2017) Squeezing out all the value of loaded data: an out-of-core graph processing system with reduced disk I/O. In: USENIX ATC, pp 125–137
28. Jun S-W, Wright A, Zhang S, Xu S (2018) Using accelerated flash storage for external graph analytics. In: ISCA. IEEE, GraFBoost
29. Jin-zhong L, Peng-jie T, Jie-wu X et al (2015) Advances in iterative MapReduce. Comput Eng Appl 51(12):123–132
30. Bu Y, How B, Balazinska M et al (2012) The HaLoop approach to large scale iterative data analysis. VLDB J 21(2):169–190
31. Bu Y, How B, Balazinska M et al (2010) HaLoop: efficient iterative data processing on large clusters. Proc VLDB Endow 3(1):285–296

32. Ekanayake J, Li H, Zhang B et al (2010) Twister: a runtime for iterative Mapreduce. In: Proceedings of the 19th ACM international symposium on high performance distributed computing, pp 810–818
33. Zhang Y, Gao Q, Gao L et al (2012) iMapReduce: a distributed computing framework for iterative computation. J Grid Comput 10(1):47–68
34. Zhang Y, Gao Q, Gao L et al (2013) PrIter: a distributed framework for prioritizing iterative computations. IEEE Trans Parallel Distrib Syst 24(9):1884–1893
35. Kang U, Tsourakakis CE, Faloutsos C (2009) Pegasus: a petascale graph mining system implementation and observations. In: Proceedings of the Ninth IEEE International Conference on Data Mining, IEEE, pp 229–238
36. Chen R, Weng X, He B et al (2010) Large graph processing in the cloud. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM, pp 1123–1126
37. Ceze L, Tuck J, Montesinos P et al (2007) BulkSC: bulk enforcement of sequential consistency. In: Proceedings of the 34th annual international symposium on computer architecture, pp 278–289
38. Shun J, Blelloch GE (2013) Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN symposium on principles and practice of parallel programming, ACM New York, NY, USA, pp 135–146
39. Han TD, Abdelrahman TS (2011) hi CUDA: high-level GPGPU programming. IEEE Trans Parallel Distrib Syst 22(1):78–90
40. Harris M (2005) GPGPU: general-purpose computation on GPUs. In: SIGGRAPH 2005 GPGPU COURSE. http://www.gpgpu.org/s2005/
41. Lee S, Min S, Eigenmann R (2009) Open MP to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN symposium on principles and practice of parallel programming, pp 101–110
42. Harish P, Narayanan PJ (2007) Accelerating large graph algorithms on the GPU using CUDA. In: Proceedings of the 14th International Conference on High Performance Computing, pp 197–208
43. Luo L, Wong M, Hwu W (2010) An effective GPU implementation of breadth-first search. In: Proceedings of the 47th Design Automation Conference, pp 52–55
44. Katz GJ, Kider Jr JT (2008) All-pairs shortest-paths for large graphs on the GPU. In: Proceedings of the 23rd ACM SIGGRAPH symposium on graphics hardware, pp 47–55
45. Hong S, Oguntebi T, Olukotun K (2011) Efficient parallel graph exploration on multi-core CPU and GPU. In: Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques, ACM New York, NY, USA, pp 78–88
46. https://www.wusiwei.com/post-2085.html
47. Robinson I, Webber J, Eifrem E (2015) Graph databases: new opportunities for connected data. O'Reilly Media Inc., Sebastopol
48. Zhong J, He B (2012) An overview of medusa: simplified graph processing on GPUs. In: Proceedings of the 17th ACM SIGPLAN symposium on principles and practice of parallel programming, ACM New York, NY, USA, pp 283–284
49. Khorasani F, Vora K, Gupta R et al (2014) CuSha: vertex-centric graph processing on GPUs. In: Proceedings of the 23rd international symposium on high-performance parallel and distributed computing, ACM New York, NY, USA, pp 239–252
50. Lingxiao M, Zhi Y, Han C, Jilong X, Yafei D (2017) Garaph: efficient GPU-accelerated graph processing on a single machine with balanced replication. In: USENIX Annual Technical Conference (ATC'), Santa Clara, CA, USA, pp 195–207
51. Zhisong F, Michael P, Bryan T (2014) MapGraph: a high level API for fast development of high performance graph analytics on GPUs. In: Proceedings of workshop on graph data management experiences and systems (GRADES'14). ACM, New York, NY, USA, Article 2
52. http://spark.apache.org/
53. Ben-Nun T, Sutton M, Pai S et al (2017) Groute: an asynchronous multi-GPU programming model for irregular computations. In: Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming, Austin, pp 235–248
54. Sha M, Li Y, He B et al (2017) Accelerating dynamic graph analytics on GPUs. Proc VLDB Endow 11:107–120
55. Zhang JL, Li J (2018) Degree-aware hybrid graph traversal on FPGA-HMC platform. In: Proceedings of the 26th ACM/SIGDA international symposium on field-programmable gate arrays, Monterey, pp 229–238

56. Zhou SJ, Prasanna VK (2017) Accelerating graph analytics on CPU-FPGA heterogeneous platform. In: Proceedings of the 29th international symposium on computer architecture and high performance computing, Campinas, pp 137–144
57. Zhang MX, Zhuo YW, Wang C et al (2018) GraphP: reducing communication for PIM-based graph processing with efficient data partition. In: Proceedings of the 24th IEEE international symposium on high-performance computer architecture, Vienna, pp 544–557
58. Dai G, Huang T, Chi Y et al (2017) Fore-graph: exploring large-scale graph processing on multi-FPGA architecture. In: Proceedings of the 25th ACM/SIGDA international symposium on field-programmable gate arrays, Monterey, pp 217–226
59. Shi XH, Liang JL, Di S et al (2015) Optimization of asynchronous graph processing on GPU with hybrid coloring model. In: Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming, San Francisco, pp 271–272
60. Dai GH, Huang TH, Chi YZ et al (2018) GraphH: a processing-in-memory architecture for large-scale graph processing. IEEE Trans Comput Aided Des Integr Circuits Syst. https://doi.org/10.1109/TCAD.2018.2821565
61. Kang U, Tong H, Sun J et al (2012) GBASE: an efficient analysis platform for large graphs. VLDB J 21(5):637–650
62. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111
63. Tasci S, Demirbas M (2013) Giraphx: parallel yet serializable largescale graph processing. In: Proceedings of European Conference on Parallel Processing. Springer, Berlin, pp 458–469
64. Khayyat Z, Awara K, Alonazi A et al (2013) Mizan: a system for dynamic load balancing in large-scale graph processing. In: Proceedings of the 8th ACM European Conference on Computer Systems. ACM, pp 169–182
65. Yan D, Cheng J, Lu Y et al (2015) Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of the 24th International Conference on World Wide Web. ACM, pp 1307–1317
66. Bao NT, Suzumura T (2013) Towards highly scalable pregel based graph processing platform with x10. In: Proceedings of the 22nd International Conference on World Wide Web Companion, International World Wide Web Conferences Steering Committee, pp 501–508
67. Donald N, Andrew L, Keshav P (2013) A lightweight infrastructure for graph analytics. In: Proceedings of the twenty-fourth symposium on operating systems principles (SOSP'13), ACM, pp 456–471
68. Zhang K, Chen R, Chen H (2015) NUMA-aware graph-structured analytics. In: Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming, PPoPP, pp 183–193
69. Abdullah G, Beltrao CL, Elizeu S-N, Matei R (2012) A yoke of oxen and a thousand chickens for heavy lifting graph processing. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12). ACM, New York, NY, USA, pp 345–354
70. Brandes U (2001) A faster algorithm for betweenness centrality. J Math Sociol 25(2):163–177
71. Broder A, Kumar R, Maghoul F, Raghavan P, Rajagopalan S, Stata R, Tomkins A, Wiener J (2000) Graph structure in the web. Comput Netw 33(1):309–320
72. Su BY, Keutzer K (2012) clSpMV: a cross-platform OpenCL SpMV framework on GPUs. In: Proceedings of the 26th ACM International Conference on Supercomputing, ACM, pp 353–364
73. Backstrom L, Huttenlocher D, Kleinberg J, Lan X (2006) Group formation in large social networks: membership, growth, and evolution. In: Proceedings of KDD, pp 44–54
74. Kwak H, Lee C, Park H, Moon S (2010) What is Twitter, a social network or a news media? In: Proceedings of WWW, pp 591–600
75. Boldi P, Rosa M, Santini M, Vigna S (2011) Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: Proceedings of WWW, pp 587–596
76. Yahoo: Yahoo WebScope (2002) Yahoo! altavista web page hyperlink connectivity graph. https://webscope.sandbox.yahoo.com/