Multimed Tools Appl (2019) 78:457–475
https://doi.org/10.1007/s11042-017-5102-2

CrossMark

GMR: graph-compatible MapReduce programming model

Weidong Zhang[1] · Boxin He[1] · Yifeng Chen[1] · Qifei Zhang[2]

Received: 23 June 2017 / Revised: 5 August 2017 / Accepted: 9 August 2017 /
Published online: 23 August 2017
© Springer Science+Business Media, LLC 2017, Corrected publication October/2017

Abstract The MapReduce programming model is widely used to parallelize data processing over the large scale of commodity computer clusters. However, on account of its monotonous data representation, it fails to express graph-parallel algorithms naturally and execute them efficiently. Alternatively, Pregel and PowerGraph could address these challenges. But they require users to familiarize another set of programming patterns and platforms, and at the same time the legacy MapReduce code also becomes incompatible and useless. In this paper, we proposed the Graph-compatible MapReduce (GMR) as an extension of Google's Standard MapReduce (SMR). In this way, graph-parallel algorithm will be naturally expressed without compromising the efficiency and simplicity, and meanwhile the conventional MapReduce programming pattern be preserved. Also, users could gain the convenience of "Think like a vertex". Based on the experimental studying, we analyzed the ratio of the redundant computation, transmission and data caching introduced in naive iterative MapReduce platforms (e.g., HaLoop, Twister). Furthermore, we discussed the difference between GMR and the graph-targeted frameworks. The evaluation experiment results show that GMR outperforms GraphX in a series of real-world graph-parallel algorithms.

✉ Weidong Zhang
zhangwd@pku.edu.cn

Boxin He
heboxin@pku.edu.cn

Yifeng Chen
cyf@pku.edu.cn

Qifei Zhang
zhangqf@cst.zju.edu.cn

[1] Peking University, Beijing, China

[2] Zhejiang University, Hangzhou, China

Keywords Distributed systems · Parallel architectures · Graph theory · Systems programs and utilities · Performance analysis and design aids · Concurrent programming · Modes of computation · Performance of systems

1 Introduction

Although MapReduce programming model is notorious on processing interdependent data, 80% of all the data in Google is still processed by MapReduce while the remaining 20% completed by Pregel [16, 17]. Two frameworks imply two sets of programming interfaces, which not only steepens the learning curve, but also causes incompatible codes.

Meanwhile, an increasing proportion of real-world data is structured as interdependent, such as the Web graph and various social networks. To benefit users with a unified framework which is compatible with independent and interdependent data, a plethora of solutions are proposed. Representative frameworks including HaLoop [2] and Twister [6] reuse the MapReduce model and add build-in support for iterative computation. Beyond that, the Spark provides the capability of graph computing after extending the component of GraphX [25].

By recognizing iterative algorithms from chained MapReduce [22] to iterative structures, users likely obtain a significant speedup. Impressively, it cuts down the overhead of I/O dramatically. In the example of the improved Hadoop, such as HaLoop and Twister, they generally achieve 2x–10x speedup [10] over original Hadoop. However, compared with the 100x speedup of GraphX [10], the naive reorganization is less than satisfying. In addition, after probing into the process of HaLoop and Twister, we have identified lots of redundant computation, transmission and caching. Experimental evaluation also shows that the ratio of repeated computation could reach up to 99.9% in some extreme scenarios. For example, in solving the problem of single source shortest path algorithm, only few vertices are active to pass the shortest path which needs all vertices to be processed in each superstep. Unfortunately, it is challengeable to filter the redundant intermediate results and messages for them[22] because no directive information is provided for runtime to discriminate the key/value pairs.

On the other hand, although the existing graph-parallel frameworks, such as Pregel and GraphX, could express and parallelize the graph algorithms naturally and efficiently, it is difficult to compose all these abstractions for each framework presents a slightly different view of graph presentation and relies on separated runtimes.

In order to enable frameworks based on GMR to express and execute graph-parallel algorithms naturally and efficiently, we extend its type mechanism by introducing a new distributed data type named as **Mutable Distributed Mirror (MDM)**. MDM is a context-related data type (vertex's context means its neighboring vertices and edges). Using MDM, programmers could naturally express the interdependency of data, such as the relationship between vertices in graph. On top of that, users could program graph-parallel algorithm with vertex-centric. The advantage of fine-grained vertex type is that it provides the directive information for MapReduce engine to filter the redundant computation, transmission and caching. GMR also adopts the Bulk Synchronous Parallel (BSP) [18] model which interprets computation into a series of global supersteps to support iterative computation. Therefore, the MapReduce programmers could not only express graph-parallel algorithm naturally, but also keep the MapReduce programming patterns and habits.

We describe and evaluate the design and implementation of GMR on a 32 nodes cluster and adopt the real-world applications on both natural and random datasets. Our key contributions are:

1) Generalizing the MapReduce Programming Model to make it compatible with graph-parallel computation;
2) Extending the type mechanism in MapReduce programming model;
3) Implementing a general computing framework which is compatible with unstructured and graph structured data under MapReduce Programming Model;
4) Proposing the concept of approximate graph partitioning and introducing the LSH-based approximate graph partitioning approach.

The rest of the paper is structured as follows. Section 2 describes the backgrounds of the state-of-the-art parallel computing models. Section 3 illustrates GRM abstraction and implementation. The applications and evaluations are enumerated in Section 4. Finally, we discuss related work and future directions.

2 Graph-parallel abstraction

2.1 MapReduce abstraction

MapReduce is a distributed programming model for large-scale data processing over massive commodity computers. Algorithm implemented with MapReduce includes several fundamental operations, parts of which are implemented by users. The fundamental operations include Map, Combine, Reduce, Shuffle and Sort. The behaviors of the operations are summarized as follows briefly.

- "Map": performs transformations (e.g., filtering, sorting or key/value pair transformation).
- "Reduce": performs summary operations (e.g., counting the number of the key/value pairs sharing the same key).
- "Combine": performs local aggregation on the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer.
- "Shuffle": redistributes key/value pairs so that these key/value pairs with the same key can be assigned to the same worker processor.
- "Sort": performs data sorting based on the key field.

The popular frameworks based on MapReduce abstraction mainly aim at the data-intensive applications and therefore are more focused on optimizing the reliability and scalability. And because the MapReduce model derives from functional programming, they were born with three limitations:

1) Unsuitable for iterative computing. Most of the iterative algorithms are organized as chained MapReduce invocations, which introduce substantial I/O overhead.
2) Inefficient message Passing. To realize message exchange between data items, MapReduce needs to generate, sort and aggregate intermediate key/value pairs according to the same key.
3) No "state" information in data item to record graph state across iterations. Every Mapper needs to explicitly convey a portion of graph to the Reducer.

The graph-parallel algorithms are usually organized as chained MapReduce invocations. But the chained MapReduce invocations show no benefit on programming with vertex-centric perspective. Furthermore the user interfaces and data representation are incapable of formulating the iterative and stateful computing process. Therefore, the original MapReduce could hardly express the graph algorithm naturally and intuitively, even cumbersomely and inefficiently.

To illustrate the chained MapReduce invocations, we adopt the example of the second degree relationship recommendation algorithm. As is shown in Algorithm 1, the example implementation looks embarrassingly counterintuitive and lengthy. In addition, the string operations also hinders the high-efficiency execution.

Algorithm 1 The second degree relationship recommendation algorithm implemented with chained MapReduce invocations

```
1:  Class Mapper_1:
2:  procedure MAP(id predecessor, id follower)
3:      Emit(id predecessor, value “pre_” + follower)
4:      Emit(id follower, value “fol_” + predecessor)
5:  Class Reducer_1:
6:  procedure REDUCE(id m, value [p1, p2, ...])
7:      for p in [p1, p2, ...] do
8:          str[] = p.split(“_”)
9:          if str[0].startWith(“pre”) then
10:             predecessors.push_back(str[1])
11:         else
12:             followers.push_back(str[1])
13:     Emit(id m, pair(predecessors, followers))
14: Class Mapper_2:
15: procedure MAP(id m, pair values)
16:     for p in values.first do
17:         for q in values.second do
18:             Emit(pair(p, q), id m)
19: Class Reducer_2:
20: procedure REDUCE(pair m, id [p1, p2, ...])
21:     for p in [p1, p2, ...] do
22:         intermediate + = “,” + p
23:     Emit(id m, value intermediate)
```

2.2 Pregel

MapReduce programming model naturally evolves from functional programming language, and the chained MapReduce invocation requires passing the entire graph from one stage to the next, which in general requires much more communication and serialization. Pregel, however, is an implementation of another different programming paradigm. It invokes vertex-program on all the active vertices in a sequence of supersteps. In Pregel, every vertex in graph takes a state flag. With the state flag, a vertex deactivates itself by voting to halt. Being inactive means the vertex has no further work to do unless it is triggered externally.

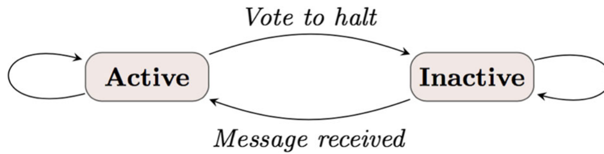

Fig. 1 Vertex state machine of pregel [17]

Reactivation can be triggered by message. When vertex is reactivated, the Pregel framework will invoke the vertex-program again. The progress can be expressed as Fig. 1.

Pregel adopts a pure message passing model instead of Remote Procedure Call (RPC) for performance [17]. A vertex V in superstep S can read messages sent to it in superstep S $-$ 1, send messages to other vertices which will be received at superstep S $+$ 1, and modify the state of V according to the execution of user interface compute().

The computation in Pregel consists of a sequence of iterations, which are named as supersteps. One superstep needs to complete an execution of vertex-programs and a global synchronization. Writing a Pregel vertex-program involves subclassing the build-in Vertex class and overriding the virtual method of compute(), which is executed at each of active vertex in every superstep. Pregel allows compute() method to access the values and edges on current vertex. What's more, vertices could also exchange messages with each other through the message passing APIs (e.g., SendMesssageTo()). Figure 2a shows the implementation of PageRank algorithm in Pregel.

2.3 GraphLab

GraphLab is an asynchronous distributed shared-memory model. By eliminating message and providing a high-level data representation of the data graph, it insulates user from the complexities of synchronization, data consistency, data races and deadlocks. Using the representation of graph, vertex-program could access values of the current vertex, adjacent vertices and adjacent edges directly. In addition, it maintains the data consistency with the configurable consistency models automatically.

Pregel

```
void Compute(MsgIterator* msgs) {
  if (superstep() >= 1) {
    double sum = 0;
    for (; !msgs->Done();
           msgs->Next())
      sum += msgs->Value();
      *MutableValue() =
             0.15 / NumVertices()
             + 0.85 * sum;
  }
  if (superstep() < 30) {
    const int64 n =
      GetOutEdgeIterator().size();
    SendMessageToAllNeighbors(
      GetValue() / n);
  } else {
      VoteToHalt();
  }
}
```

(a)

GraphLab

```
void GLPageRank(vertex v) {
  float accum = 0;
  foreach (nbr in v.predecessors) :
    accum += nbr.val /
              v.predecessors.size();
  v.val = 0.15 + 0.85 * accum;
}
```

(b)

PowerGraph

```
gather(Du, D(u,v), Dv):
  return Dv.rank / #outNbrs(v)
sum(a, b): return a + b
apply(Du, acc):
  rnew = 0.15 + 0.85 * acc
  Du.delta = (rnew - Du.rank) /
  Du.rank = rnew
scatter(Du ,D(u,v) ,Dv ):
  if(|Du.delta|> ) Activate(v)
  return delta
```

(c)

Fig. 2 The PageRank algorithm implemented in Pregel, GraphLab and PowerGraph

In GraphLab, the vertex-program is triggered by its neighboring vertex-programs in the future, which means vertex-program will prevent neighboring programs from running simultaneously to ensure the serializability of executions of vertex-programs. In Fig. 2b, we implement PageRank algorithm to demonstrate that vertex-program operates neighboring vertex values directly.

2.4 PowerGraph

PowerGraph is another graph-parallel computation abstraction concentrating on the graphs with highly skewed power-law degree distributions.

It adopts the GAS (Gather, Apply, Scatter) model to express vertex program. Computation in the PowerGraph needs implementing the GASVertexProgram interface, and explicitly decomposing computation into the gather, sum, apply and scatter functions. The gather function is invoked in parallel on the neighboring edges of vertex, and returns temporary accumulator, which will be combined by using the commutative and associative sum operation. After the completion of gather phase, the apply function takes the accumulator and computes the new state of current vertex. Finally, the scatter function is invoked in parallel on the neighboring edges of current vertex to compute new edge values and write back to the graph. Three barriers are imposed between the functions to synchronize data. Figure 2c demonstrates the example code implemented in PowerGraph.

PowerGraph combines the feathers of Pregel and GraphLab. From Pregel, PowerGraph borrows the associative and communicative gather operation [10]. From GraphLab, PowerGraph takes the data representation view of data graph to eliminate the need for user to formulate the movement of information.

2.5 GraphX

GraphX is a component of Spark, targeting at graphs and graph-parallel computation. It extends the Spark RDD by introducing a new Graph abstraction [25]: a directed multigraph with properties attached to each vertex and edge. In order to simplify graph construction and transformation, GraphX offers a set of primitive operations (e.g., degrees, subgraph, joinVertices and aggregateMessages) and a variant of Pregel. Using these primitives, GraphX could implement the PowerGraph and Pregel abstractions in less than 20 lines of code [25]. In addition, more and more open source implementations of graph algorithms and builders are added in GraphX to simplify graph analytics tasks.

3 GMR model

Although plenty of studies have come to a conclusion that the MapReduce is not suitable for computation with much interdependency. Paradoxically, we find that it does fit for the computation with much interdependency after necessary extending. Therefore, the motivation of this work is to enhance the MapReduce programming model to support graph-parallel algorithms naturally and efficiently.

In functional programming, the higher-order function **map(f(x), list)** is used to generate and return a new list by applying the **f(x)** to each list item, and **reduce(g(x), list)** is used to aggregate the items in list with aggregation function **g(x)**. SMR borrows the concept of map/reduce in functional programming, but limits the data type of list elements to key/value pair.

After the above simplification, the core role of the MapReduce framework is to provide a runtime that enables the execution of map/reduce in parallel over the unreliable commodity PCs. Via this methodology, the parallelization is highly achieved. However, its drawback is obvious as well, especially the constrain on the type, which brings in considerable overhead during data exchange.

The frameworks based on SMR require all the data to be presented as key/value pairs, including the input, intermediate and final results. But the key/value pair is incompetent to express graph naturally, even damages the structure information of graph. Because of these constraints, it is hard to exchange the messages between vertices and control state of vertices. In order to bypass these defects, prior frameworks based on SMR need to introduce lots of redundant global shuffling, sorting, grouping, aggregating, communication and caching.

From a fresh perspective, GMR extends MapReduce on two aspects. One is the type system; the other is the execution engine. With such extensions, GMR could support user to program graph-parallel algorithms with vertex-centric perspective and meanwhile keep the familiar programming pattern of MapReduce.

3.1 "Think-like-a-vertex"

Using vertex-centric programming model, user could express graph algorithm with "Think-like-a-vertex". Vertex-program describes graph algorithm from a single vertex's perspective and applys on each vertex of graph for a loosely coupled execution in parallel [17]. The execution of vertex-program frequently needs collecting/scattering messages among peer vertices. Existing frameworks provide the following ways to implement that.

Pregel, GraphLab and PowerGraph all provide the interfaces of defining a vertex-centric program but in slightly different ways. In Pregel, vertex-program is implemented in compute(). Message passing API enables compute() to gather and scatter with other vertices between supersteps. PowerGraph requires implementing three interfaces to express vertex-program actions, which include gather, computation and scatter.

To leverage vertex-centric programming model, GMR also provides the "Think-like-a-vertex" philosophy for user. But with slightly different interface, the operation of vertex-program is implemented in map/reduce. Direct data accessing between neighbors is provided like in GraphLab and PowerGraph. In addition, more fundamental functions are provided to control the structures of vertex and message in GMR.

3.2 Graph presentation

The vertex-centric approach needs users to focus on processing each vertex independently. Then system composes these independent executions to lift computation to a graph. However, the computation applied on each vertex usually needs obtaining information from others. For example, a broad set of graph algorithms require gathering information from predecessors and scattering messages to successors. Pregel provides message passing API to achieve that. And PowerGraph factors user user-program into three phases, in which gather() and scatter() appear.

In GMR, the manner of data accessing is closely analogous to GraphLab. The vertex-program (i.e., map/reduce) could not only operate the values on current vertex passed by parameters, but also could access the values on adjacent edges and adjacent vertices directly. GMR stores the values on a new designed distributed data structure, which is described in the following part of this section in detail.

3.2.1 Type extension of graph representation

To realize direct data accessing between neighbors, GMR caches vertex's adjacent vertices residing on different processors. To minimize the memory footprint of data cache, user is enabled to customize which part of vertex can be cached, accessed and available by whom (predecessor or successor). To describe these functions and operations formally, we abstract and encapsulate a new data type, **Mutable Distributed Mirror (MDM)**. MDM owns the following three properties:

- Data is distributed over different processors/machines;
- Data mirror is adopted to facilitate direct accessing between neighbors residing on different machines;
- Data structure of mirror is mutable, and can be tailored from its master record by user;

With the concept of MDM, we design the Vertex class. The Vertex class is a new build-in system class, which derives from key/value pair. It contains several basic properties to describe a graph vertex but not limited to vertex. The UML diagram of KeyValue class and Vertex class is shown in Fig. 3.

Implementing a graph algorithm in GMR, user first needs to define a new class which inherits from Vertex class, declares fields and then customizes the accessibility of these fields with annotations provided by the system. If field is annotated as accessible from predecessor (resp. successor), it can be accessed by all its predecessors (resp. successors). The annotations are similar to that of Java. If fields are annotated, it will be sent to and cached in corresponding machines when its value is updated.

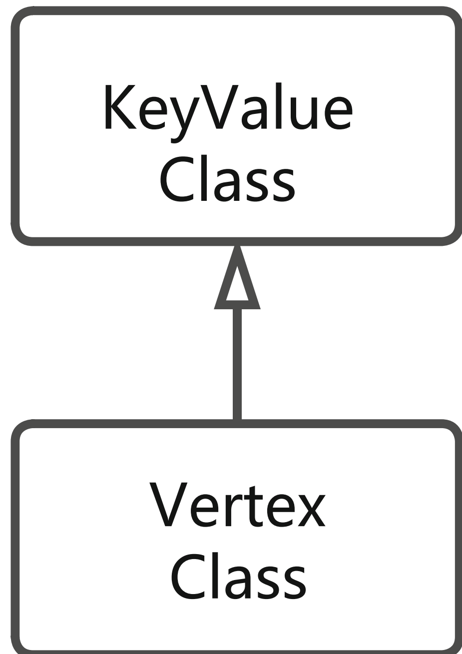

Fig. 3 The class hierarchy of KeyValue class and vertex class, vertex inherits from KeyValue

For ease of use, three member variables are introduced in Vertex class to reference values of its neighbors, i.e., predecessor, successor and nbrs, which denote predecessors, successors and all neighbors respectively. Annotations and their corresponding referencing variables are shown in Table 1.

Therefore, fields annotated as:

```
@accessible(value = predecessor / successor)
DataType fieldname;
```

could be accessed in vertex-program as:

```
v.successor[i].filedname or
v.predecessor[i].fieldname or
v.nbrs[i].fieldname
```

The i in the above expressions indicates the index of its predecessor or successor.

With the new data type, GMR simplifies the programming of graph-parallel algorithms significantly, such as the single source shortest path (SSSP) [4] and second degree relationship recommendation algorithm. For SSSP, each vertex could directly read the shortest path from its all predecessors. As to the second degree relationship recommendation algorithm, vertex-program needs to gather values from both predecessors and successors simultaneously, it can read directly now. More real-world applications of Vertex class are presented in Section 3.3.

What's more, user could set the execution state of vertex by an identifier in Vertex class. The state identifier is also used by runtime to determine whether to execute vertex-program on current vertex or not. If state appears active, runtime will apply the vertex-program on current vertex; otherwise, runtime will ignore the current vertex until it is reactivated again. To demonstrate the predefined subclass of Vertex class, Fig. 4 shows some basic example classes. In Fig. 4a, the **KMeansType** is used to represent the input, intermediate and result data type. Because the KMeans clustering algorithm is not a graph-parallel algorithm, **KMeansType** inherits from the **KeyValue** class and no field is annotated. In Fig. 4b, the **PRVertex** is used to represent the webpage in PageRank algorithm, which inherits from **Vertex** class. The **SSSPVertex** in Fig. 4c stands for the graph vertex in SSSP algorithm.

To deliver these mechanisms efficiently, we design and extend the underlying graph storage and partitioning algorithm, which will be described in the following two sections.

3.2.2 The underlying representation of graph and message

GMR mainly employs two measures to minimize the in-memory presentations of graph and message to support a larger scale of computation.

Table 1 Annotations and corresponding access interfaces

Annotation	Access interface
@accessible(value=predecessor)	vertex.predecessor/nbrs
@accessible(value=successor)	vertex.successor/nbs

(a) KMeans:

```
class KMeansType : KeyValue
int centerId
vector⟨int⟩ value(32)
```

(b) PageRank:

```
class PRVertex : Vertex
int id
@Accessible(value = successor)
double value
@Accessible(value = successor)
int nhrSize
```

(c) SSSP:

```
class SSSPVertex : Vertex
int id
@Accessible(value = successor)
double value
```

Fig. 4 Example of user defined data types in GMR

One is that GMR adopts Compressed Sparse Row (CSR) [3] to store graph in memory. CSR represents a matrix M with three arrays (one-dimensional), i.e., nonzero values, the extents of rows, and column indices. According to our design of data accessing model, we add three fields to CSR structure, i.e., predecessor, successor and nbrs.

The other is partial mirror. Partial mirror is used to cache adjacent vertices, which reside on different processors, to support direct data access as well as minimize mirror and message to a reasonable size. The implementation of partial mirror is simple and intuitive, if field is annotated as @accessible(value = predecessor / successor) then the value of this field will be passed to and cached in corresponding predecessor[i] or successors[i]. The non-annotated fields will not be delivered or cached. So the mirror in GMR is termed as a partial mirror instead of a full copy of adjacent vertex. When no field is annotated, the referencing arrays (predecessor, successor, nbrs) will be set to empty. The data consistency between mirror and its master is ensured by the BSP model. Figure 5 demonstrates memory images of GMR model and PowerGraph model. In Fig. 5a, the dented pizzas imply they are partial mirrors.

With the supports of annotation mechanism and partial mirror, GMR could drastically reduce the volume of communication and cached data, as well as simplify the data operations.

3.2.3 Graph partition

In mathematics, the Graph partition problem is to partition vertices of a graph into multiple approximately equal-sized sets so that the number of cut-edge between subgraphs is minimum. It is an important NP-complete problem with applications on VLSI CAD, processor allocation, and many other areas [5, 8, 9, 14, 15, 19–21, 23, 27–29]. Although there has

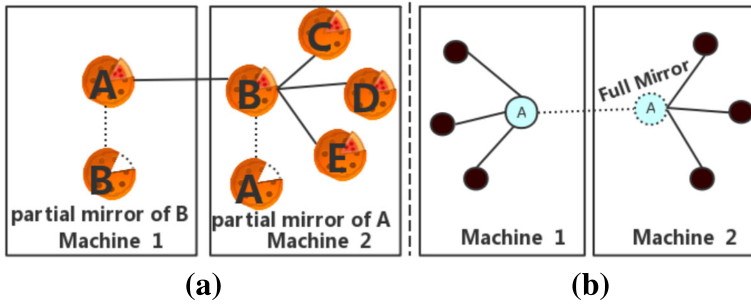

Fig. 5 Memory image of vertex in GMR and PowerGraph, the mirror in PowerGraph is a full copy of its master vertex, but in GMR it is a partial copy of its master vertex

been a lot of graph partitioning algorithms, the dominant graph computing frameworks still adopt random partitioning category, such as in Pregel and GraphLab.

The major shortcomings of existing algorithms are their high complexity and lack of distributedness. In this part, we present a new approximate partitioning algorithm, **Locality-sensitive Hashing based graph partitioning (LSHGP)**.

Locality-sensitive Hashing (LSH) is widely used in approximate searching in big dataset, which mainly depends on the similarities between data items. LSH could hash the similar items to neighboring regions in hash tables. Our casual experiment results show that adjacent vertices (and edges) have analogue formal representations. Such as the graph in Fig. 6, the expressions of the vertices and their similarities are shown in Table 2. Theoretically, higher similarity vertices should be hashed to closer regions in hash table with greater probability [16].

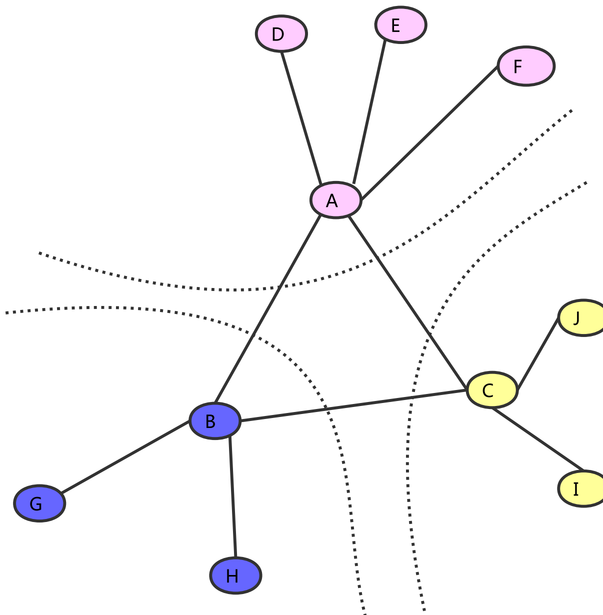

Fig. 6 Demo graph

Table 2 Similarities between adjacent vertices

VID	Vertex represent	Jaccard similarity									
		A	B	C	D	E	F	G	H	I	J
A	$\langle A, B, C, D, E, F\rangle$	1	.37	.37	.33	33	.33	.14	.14	.14	.14
B	$\langle A, B, C, G, H\rangle$	.37	1	.42	.16	.16	.16	.4	.4	.16	.16
C	$\langle A, B, C, I, J\rangle$	.37	.42	1	.16	.16	.16	.16	.16	.4	.4
D	$\langle A, D\rangle$	.33	.16	.16	1	.33	.33	0	0	0	0
E	$\langle A, E\rangle$	.33	.16	.16	.33	1	.33	0	0	0	0
F	$\langle A, F\rangle$	.14	.16	.16	.33	.33	1	0	0	0	0
G	$\langle B, G\rangle$	.14	.4	.16	0	0	0	1	.33	0	0
H	$\langle B, H\rangle$	.14	.4	.16	0	0	0	.33	1	0	0
I	$\langle C, I\rangle$	.14	.16	.4	0	0	0	0	0	1	.33
J	$\langle C, J\rangle$	.14	.16	.4	0	0	0	0	0	.33	1

Depending on the LSH and the similarity between vertices, adjacent vertices can be hashed to neighboring regions with a certain probability. Then by appropriate segmentation on hash table, the equal sized blocks of vertices can be regarded as approximately subgraphs after partition.

Compared with existing algorithms, such as Metis [12], Multilevel k-way partitioning scheme [13], spectral graph partitioning algorithm [11], etc., the outstanding advantage of LSHGP is the low complexity, which is quite close to that of random graph partitioning algorithm.

Table 3 demonstrates the improvement on the number of cut-edges between partitions and time compromise. We could generally achieve about 20 to 100% improvement on the number of cut-edge. But the result is far from our expectation, which we think attributes to the locality-sensitive algorithm and the structure of our hash table. After this work, we will do more research on this issue.

3.2.4 More applications of vertex class

As is shown in Fig. 7, when algorithm is implemented by SMR, intermediate results need to be shuffled and sorted among all the processors regardless of data structure and content, such as in Hadoop, HaLoop, Twister, etc. This could impose considerably overhead on local I/O, network communication and processing speed, particularly to these iterative algorithms.

Table 3 Comparison between random graph partition and LSH-base graph partition

(#vertices, #edges)	Random GP		LSH-based GP	
	#cut-edge	time	#cut-edge	time
G4elt(766, 1314)	1,787	0.1s	533	0.3s
GMesh(258k, 513k)	361,083	1.5s	229,106	3.9s
GSOC(4.8m, 68m)	6,092,367	138s	5,197,754	229s

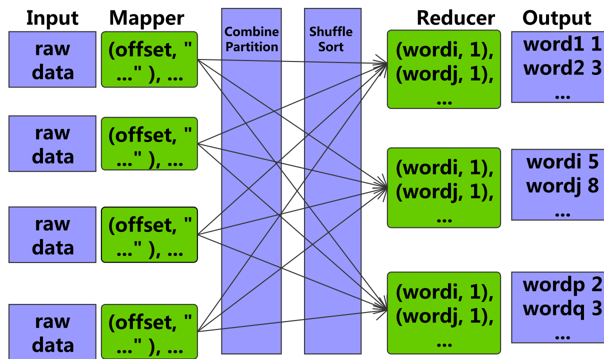

Fig. 7 SMR execution procedures

Actually, lots of intermediate results do not need to be migrated, such as the vertices in graph computing. Researchers have explored and tried to address this challenge. A. Elgohary proposed the Stateful MapReduce [7], which contains an additional state identifier to conduct runtime to distinguish the expired and fresh data. Its data exchange is inherently based on shuffling and sorting, so the redundant task still exists. Y. Zhang and Q. Gao etc. put forward the iMapReduce [30] providing two different sets of API to operate and manage different types of data (i.e., static data and state data). In GMR, we address these problems by extending the type mechanism and introducing new programming model. Besides the graph computing, we have found the new type mechanism and programming model can be used in more fields and applications.

The direct use of Vertex class is to express the vertex in graph. Surprisingly, it could be used to express much more complicated objects, such as the type of catalyst [24] and enzyme [1] in the simulation of biochemical reactions. Both catalyst and enzyme keep unchanged and transform other inputs into other forms concurrently during reactions. Figure 8a displays the execution process of PageRank algorithm, in which the balls inherit from Vertex class and represent webpages, and the coins indicate the key/value pairs delivering the shared PageRank values. During the execution of PageRank, instance of PRVertex keeps still on the original processor, emits and receives key/value pairs. Figure 8b demonstrates that the Vertex class is used to represent catalytic in biochemical reactions. Besides the webpage and catalytic mentioned above, we could imagine that more entities could be abstracted as Vertex, such as planet in astrophysics and so on.

3.3 Case study

GMR enables users to program with vertex-centric. Compared with Pregel and PowerGraph, the map/reduce function is reused to implement vertex-program rather than introducing another new set of user interfaces.

When writing GMR vertex-program, user needs to override the methods of map() and reduce(). In these methods, user could access relevant values directly through the interfaces defined in Vertex class. A comparison among PageRank implementations by Pregel, GraphLab, PowerGraph and GMR is shown in Figs. 9a and 2a, b, c.

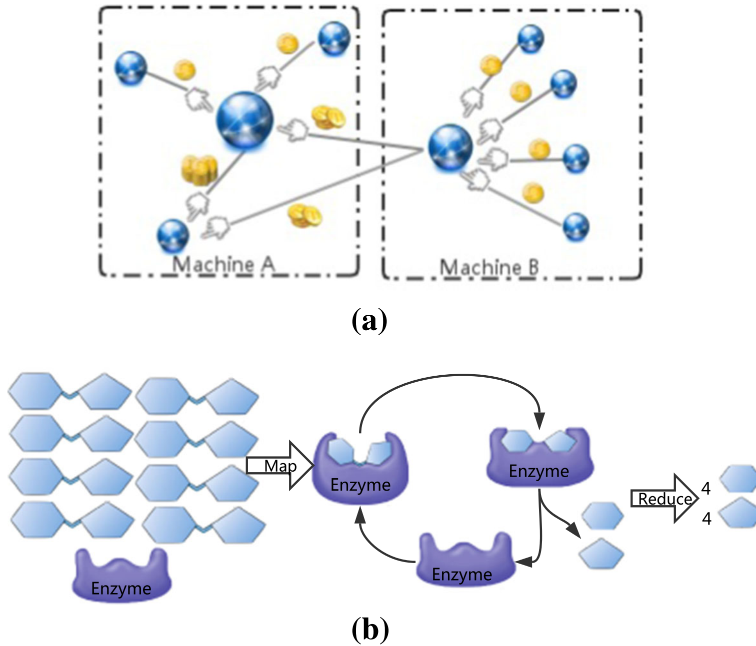

Fig. 8 Diagram of Vertex usages. **a** Vertex class used in PageRank algorithm, **b** Enzyme class used in simulation of biochemical reaction

Similar to SMR, map() is mainly responsible for transformation operations and reduce() primarily performs summary operations. In the example of PageRank, map() generates the shared value messages, while reduce() aggregates and sums up the messages.

3.4 Iterative MapReduce engine

To cut down the redundant I/O operation, GMR adopts the BSP model to support iterative computation. Using BSP model, the intermediate results will no longer be shifted back and forth between disk and memory. The movement direction of intermediate results are shown in Fig. 10.

Meanwhile, to support multiple types running in one runtime simultaneously, we adopt the polymorphism mechanism of language to provide interfaces for user-defined types. User could customize actions of map, combine, shuffle and reduce. The system also provides default actions for superclass of Vertex class. For example instances of Vertex class will keep still on shuffle stage. During executing, the engine will invoke different actions upon different types. For instance, in shuffle stage, GMR moves key/value pairs with the same way of vertex distribution, and at the same time keeps vertices still. Figure 10 illustrates the different execution processes corresponding to different types.

To describe the execution process in detail, we sketch the flowchart of runtime in Fig. 10. At input stage, map() accepts input from both file reader module and reduce output module. After processing of map(), multiple types of output may be generated, such as Vertex and

PageRank

```
void map(Vertex v, vector<KV> kvs) {
  if (v.nbrSize == 0) return;
  float value = 0.0;
  for (int i = 0; i < v.predecceesor.size();
       ++i) {
    value += v.predecessor[i].value
         / v.predecessor[i].nbrSize;
  }
  kvs.push_back({v.id, value});
}
KV reduce(list<KV> &kvs) {
  float sum = 0.0;
  for (auto kv : kvs) {
    sum += kv.value;
  }
  sum = 0.5 * sum + (1 - 0.5) / ntxs;
  return {kvs[0].key, sum};
}
```

(a)

SSSP

```
void map(Vertex v, vector<KV> kvs) {
 kvs.push_back({v.id, v.value});
 for (int i = 0;
      i < v.predecceesor.size(); ++i) {
  kvs.push_back({v.id,
      v.predecceesor[i].value + 1.0});
 }
}
KV reduce(list<KV> kvs) {
 float shortestPath = FLT_MAX;
 for (auto kv : kvs) {
   if (kv.value < shortestPath)
     shortestPath = kv.value;
 }
 return {kvs[0].key, shortestPath};
}
```

(b)

KMeans

```
KmeansType map(KmeansType v,
  vector<vector<int>> centers) {
 int center;
 double min = DBL_MAX;
 for (int i = 0;i < centers.size(); ++i) {
  if (distance(v.value, centers[i]) < min){
    center = i;
    min = distrance(v, centers[i]);
  }
 }
 return v;
}
vector<int> reduce(KmeansTye kvs){
 vector<int> center;
 for (int i = 0; i < kvs[0].value.size();
      ++i) {
   for (int j = 0; j < kvs.size(); ++j)
     sum += kvs[j][i]
   center[i] = sum / kvs.size();
 }
 return center;
}
```

(c)

Fig. 9 The PageRank, SSSP and KMeans algorithms implemented in GMR

simple key/value pair. Then, shuffle module will move the intermediate results to different destinations according to their types. The default destination of vertex is their current positions. After reduce(), all the vertices will be checked whether they have become convergent. If any non-convergent vertex exists, the engine will start another superstep until all vertices become convergent or reach other external precondition, such as reaching specified number of iterations.

4 Applications and evaluation

GraphX is an open source implementation of Pregel on Spark, and famous for its high efficiency and vertex-centric programming. The following contrast tests will take it as benchmark. We tested and compared the performances of Spark, GraphX and GMR on several algorithms with both natural and random graphs.

These tests are conducted on a cluster of 32 8-core commodity Servers with 46GB RAM. Three datasets are used. The first one consists of 100,000,000 32-demensional random vectors; the second one is a mesh graph with 258,569 vertices and 1,026,264 edges and the last is a social network graph (SOC) [26] published by Stanford Network Analysis Project, which contains 3,997,962 vertices and 34,681,189 edges. To demonstrate the highly skewed degree distributions of the SOC dataset, we plot the fan-in and fan-out degree distribution in log scale in Fig. 11.

4.1 Single source shortest path (SSSP)

The SSSP problem requires finding the shortest paths from a source vertex to all the other vertices in graph. The quantity of iteration is the max distance to the source vertex. For simplicity and conciseness, we assume that all the distance between the adjacent vertices is 1.0. As is shown in Fig. 4c, SSSPVertex is subclass of Vertex and used to present the vertex in SSSP problem. The value in SSSPVertex indicates the shortest path value from source to the current vertex and is annotated as accessible from its successors (@accessible(value = successor)).

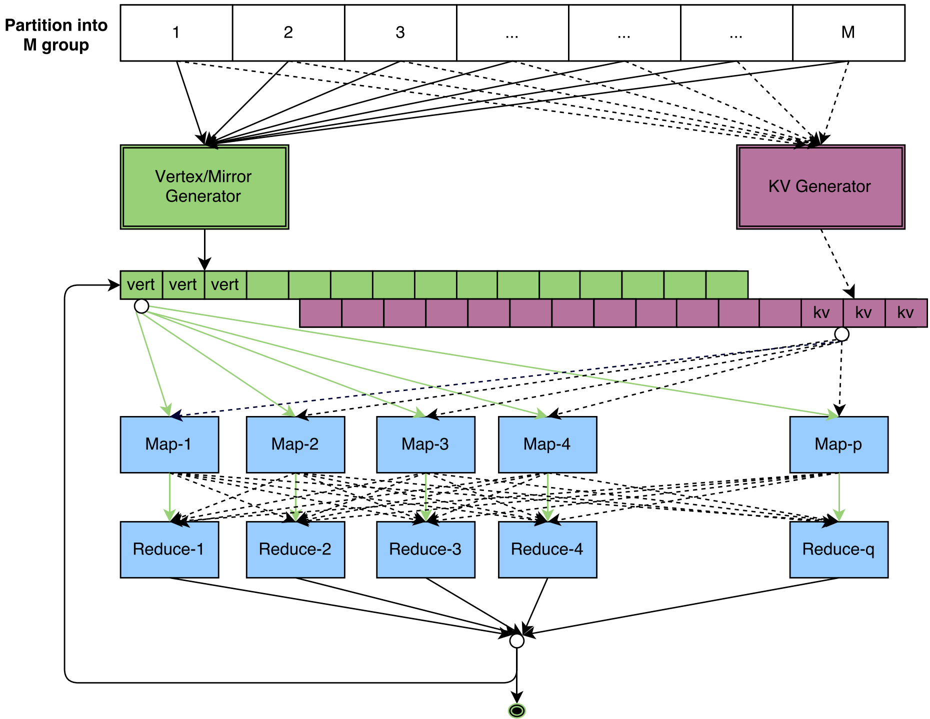

Fig. 10 Execution order of GMR with graph data and simple key/value pair

The vertex-program is shown as Fig. 9b. In map() method, it enumerates one's all predecessors to calculate and output the intermediate vertices and intermediate key/value pairs: {key, value}, where the key and value are current vertex id and predecessor's shortest path plus the weight of the edge between them, respectively. In reduce() method, it selects the pair with minimum value by group.

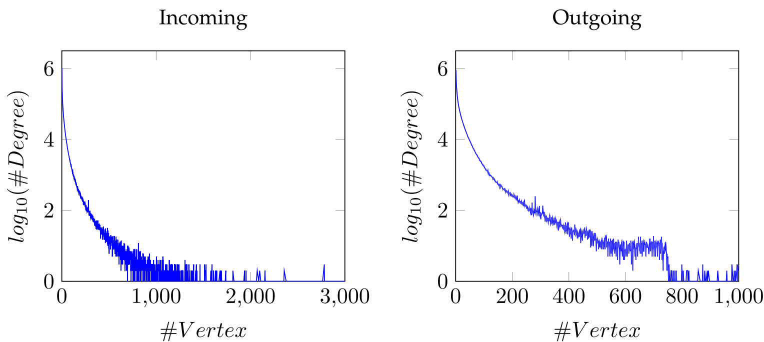

Fig. 11 In and out degree distributions in log scale

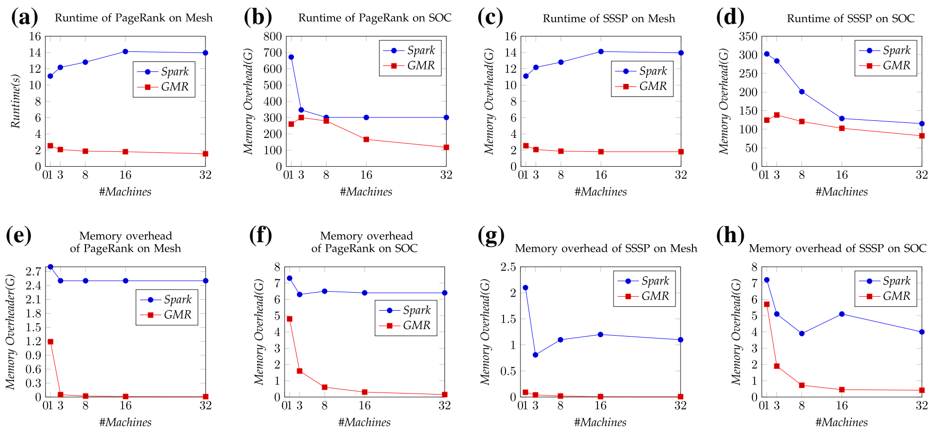

Fig. 12 Performance of GMR on speedup and memory

To synchronize the value cached in a different processors, runtime needs to synchronize the partial mirror when its master record is updated. In the implementation, we arrange the annotated fields in sequence. Figure 12a, b, e, f present the processing time and memory overhead of GraphX and GMR. Numbers show that GMR obtains better speedup than GraphX, and meanwhile takes up less memory.

4.2 PageRank

PageRank is a link analysis algorithm. It works by summing up the number and quality of links to a page to determine a rough estimate of how important the page is. A GMR implementation of a PageRank algorithm is shown in Figs. 4b and 9a. The PageRankVertex class inherits from Vertex class. The type of value in PageRankVertex is double and annotated as accessible from successors.

In map() method, it enumerates a vertex's all predecessors to calculate and output intermediate key/value pair: {key, value}, where the key is the id of current vertex and the value is the PageRank value shared to its successors. In reduce() method, it sums up the values with the same key by group with the specified formula. Figure 12c, d, g, h shows the experiment results that GMR could gain 5x-7x speedup than Spark GraphX and meanwhile takes up less memory.

4.3 KMeans clustering

Besides the above graph-parallel algorithm, we also test and evaluate the frameworks with non-graph algorithm and dataset. KMeans clustering is used to partition n samples into k clusters in which each sample belongs to the cluster with the minimum mean.

In this case, the sample data is 100,000,000 32-demensional vectors generated randomly. The map() method computes the distances of current sample to every center and output an intermediate key/value pair, where the key is the id of the closest cluster and the value is the sample data. In reduce() method, it calculates the new cluster centers with the same key in group. The code is shown in Fig. 9c. The experiment results in Fig. 13a, b show the GMR is between 100x–300x faster than Spark while taking up less memory. As to the

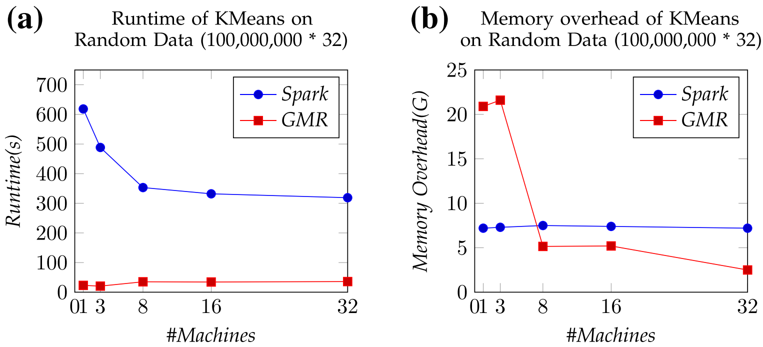

Fig. 13 The performance comparison of GMR and spark on KMeans clustering

amazing speedup effect, we think it is mainly owing to the C++ programming language and small scale of dataset, because the KMeans algorithm could be implemented by MapReduce without any extension, which is supported in both Spark and GMR.

For the three example algorithms, GMR achieves appreciable speedups. Besides the contribution of GMR programming model, the improvement partly attributes to the programming language of C++ and MPI implementation. Compared with the programming language of GraphX, C++ generally obtains better performance than Java and Scala.

5 Conclusion

In this work, we generalize the MapReduce programming model, making it capable for processing large-scale graph-parallel computation. Consequently, the MapReduce programmer no longer needs to master another dedicated graph computing technology and meanwhile the amount of legacy code becomes reusable. The scalability and fault-tolerance of GMR inherit from SMR, which could run on clusters with large number of commodity PCs and tolerant machine failures gracefully [22]. Last but not the least, the speedup of GMR has achieved comparable performance with state-of-the-art computing framework.

Like Pregel, GMR adopts the BSP model to control the data and state synchronization between supersteps. Using synchronization model means the running time depends on the quantity of iterations by linear correlation. If it is used in the algorithm with relatively long iteration path, e.g., shortest path, long running time and low utilization ratio of cluster will happen. We have started to investigate the solutions for folding iteration path, which in turn can reduce the runtime and promote cluster's usage.

When running these tests, we observed that some faster processors always needed to wait for some slower processors (straggler). Through analysis, we found that there are two major reasons responsible for this phenomenon. One is that the input workload is not balanced. The other is that processors provide the imbalanced capabilities. In response to the former reason, LSHGP will become our next research issue. We believe that the ideas and designs in GMR could impact more on the design of big data platforms.

Acknowledgments Our thanks to the Institute of Process Engineering, Chinese Academy of Science for their help. This research was supported by the Zhejiang Engineering Research Center of Intelligent Medicine(2016E10011) and the research and application of key technologies for rapid individualized sculpture manufacture and carving stone materials appraisal.

References

1. Beierlein F, Clark T (2005) Computer simulations of enzyme reaction mechanisms: simulation of protein spectra. High performance computing in science & engineering Munich 2004, Springer, pp 245-259
2. Bu Y, Howe B, Balazinska M, Ernst MD (2010) Haloop: efficient iterative data processing on large clusters. Proceedings of the Vldb endowment 3(1):285–296
3. Buluç A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE (2009) Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: SPAA '09: proceedings of the twenty-first annual symposium on parallelism in algorithms and archi, pp 233–244
4. Cherkassky BV, Goldberg AV, Radzik T (1996) Shortest path algorithms: theory and experimental evaluation. Math Program 73(2):129–174
5. Chua TS, Chua TS, Chua TS, Chua TS, Chua TS (2016) Learning from collective intelligence: Feature learning using social images and tags. ACM Trans Multimed Comput Commun Appl 13(1):1
6. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G (2010) Twister: a runtime for iterative mapreduce. In: ACM international symposium on high performance distributed computing, pp 810–818
7. Elgohary A, (2012) Stateful mapreduce
8. Gao Z, Zhang H, Xu GP, Xue YB, Hauptmann AG (2014) Multi-view discriminative and structured dictionary learning with group sparsity for human action recognition. Signal Process 112(C):83–97
9. Gao Z, Zhang H, Xu GP, Xue YB (2015) Multi-perspective and multi-modality joint representation and recognition model for 3d action recognition. Neurocomputing 151:554–564
10. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) Powergraph: distributed graph-parallel computation on natural graphs. In: Usenix conference on operating systems design and implementation, pp 17–30
11. Guattery S, Miller GL (1995) On the performance of spectral graph partitioning methods. In: ACM-SIAM symposium on discrete algorithms, pp 233–242
12. Karypis G, Kumar V (1998) Metis: a software package for partitioning unstructured graphs. In: International cryogenics monograph, pp 121–124
13. Karypis G, Kumar V (1999) Multilevel k-way partitioning scheme for irregular graphs. J Parallel Distrib Comput 48(1):96–129
14. Liu AA, Nie WZ, Gao Y, Su YT (2016) Multi-modal clique-graph matching for view-based 3d model retrieval. IEEE Trans Image Process 25(5):2103–2116
15. Liu AA, Su YT, Nie WZ, Kankanhalli M (2016) Hierarchical clustering multi-task learning for joint human action grouping and recognition. IEEE Trans Pattern Anal Mach Intell 39(1):102–114
16. Lv Q, Josephson W, Wang Z, Charikar M, Li K (2007) Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: International conference on very large data bases, University of Vienna, Austria, September, pp 950–961
17. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G (2009) Pregel: a system for large-scale graph processing. In: SPAA 2009: proceedings of the ACM symposium on parallelism in algorithms and architectures, Calgary, Alberta, Canada, August, pp 135–146
18. Miller F (1993) A library for bulk-synchronous parallel programming. In: Proceedings of the BCS parallel processing specialist group workshop on general purpose parallel computing, pp 100–108
19. Nie W, Liu A, Li W, Su Y (2016) Cross-view action recognition by cross-domain learning *. Image Vis Comput 55:109–118
20. Nie WZ, Liu AA, Gao Z, Su YT (2015) Clique-graph matching by preserving global & local structure. In: Computer vision and pattern recognition, pp 4503–4510
21. Nie WZ, Liu AA, Su YT (2016) 3D object retrieval based on sparse coding in weak supervision. J Vis Commun Image Represent 37(C):40–45
22. Raji RP (2009) Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1):107–113
23. Savage JE, Wloka MG (1991) Parallelism in graph-partitioning. J Parallel Distrib Comput 13(3):257–272
24. Weilenmann M (2012) Aspects of highly transient catalyst simulation. Catal Today 188(1):121–134
25. Xin RS, Gonzalez JE, Franklin MJ, Stoica I (2013) Graphx: a resilient distributed graph system on spark. In: International workshop on graph data management experiences and systems, pp 1–6
26. Yang J, Leskovec J (2015) Defining and evaluating network communities based on ground-truth. Knowl Inf Syst 42(1):745–754
27. Zhang H, Liu W, Liu W, He X, Luan H, Chua TS (2016) Discrete collaborative filtering. In: International ACM SIGIR conference on research and development in information retrieval, pp 325–334
28. Zhang H, Zha ZJ, Yang Y, Yan S, Chua TS (2014) Robust (semi) nonnegative graph embedding. IEEE Trans Image Process A Publ IEEE Signal Process Soc 23(7):2996
29. Zhang H, Zha ZJ, Yang Y, Yan S, Gao Y, Chua TS (2013) Attribute-augmented semantic hierarchy: towards bridging semantic gap and intention gap in image retrieval. In: Proceedings of the 21st ACM international conference on Multimedia, pp 33–42. ACM
30. Zhang Y, Gao Q, Gao L, Wang C (2012) Imapreduce: a distributed computing framework for iterative computation. J Grid Comput 10(1):1112–1121