Data Sci. Eng. (2017) 2:94–105
DOI 10.1007/s41019-017-0034-4

Graph Partitioning for Distributed Graph Processing

Makoto Onizuka[1] · **Toshimasa Fujimori**[1] · **Hiroaki Shiokawa**[2]

Received: 10 December 2016 / Accepted: 17 January 2017 / Published online: 4 February 2017

Abstract There is a large demand for distributed engines that efficiently process large-scale graph data, such as social graph and web graph. The distributed graph engines execute analysis process after partitioning input graph data and assign them to distributed computers, so the quality of graph partitioning largely affects the communication cost and load balance among computers during the analysis process. We propose an effective graph partitioning technique that achieves low communication cost and good load balance among computers at the same time. We first generate more clusters than the number of computers by extending the modularity-based clustering, and then merge those clusters into balanced-size clusters until the number of clusters becomes the number of computers by using techniques designed for graph packing problem. We implemented our technique on top of distributed graph engine, PowerGraph, and made intensive experiments. The results show that our partitioning technique reduces the communication cost so it improves the response time of graph analysis patterns. In particular, PageRank computation is 3.2 times faster at most than HDRF, the state-of-the art of streaming-based partitioning approach.

✉ Makoto Onizuka
onizuka@ist.osaka-u.ac.jp

Toshimasa Fujimori
fujimori@ist.osaka-u.ac.jp

Hiroaki Shiokawa
shiokawa@cs.tsukuba.ac.jp

[1] Graduate School of Information Science and Technology, Osaka University, 1-5, Yamadaoka, Suita, Osaka 565-0871, Japan

[2] Center of Computational Sciences, University of Tsukuba, 1-1-1, Tennoudai, Tsukuba, Ibaraki 305-8573, Japan

Keywords Graph partitioning · Graph mining · Distributed processing

1 Introduction

Large-scale graph data such as social graphs and web graphs have emerged in various domains. As an example of social graph, the number of daily active users in Facebook reached 1.13 billion on average for June 2016 an increase of 17% year-over-year reported in the Facebook reports second quarter 2016 results:[1] vertexes and edges represent users and their relationships, respectively.

To analyze such large-scale graph data efficiently, distributed graph engines have been developed and they are widely used in graph analysis field. Some examples are Pregel [1], GraphLab [2], PowerGraph [3], and GraphX [4]. Distributed graph engines commonly (1) partition input graph data into sub-graphs, (2) assign each sub-graph to each computer, and (3) make graph analysis over the distributed graph. Each computer iteratively analyzes the assigned sub-graph by updating the parameters assigned to the vertexes/edges. Notice that the sub-graph assignment to computers largely affects the communication cost and load balance during graph analysis. The commutation cost increases to the number of cross-partition vertexes/edges, because communication between different computers is required when parameters are updated by referring to adjacent vertexes/edges in remote computers. The computation cost of each computer depends on the number of vertexes/edges assigned to the computer [5], so load imbalance occurs among computers when the number of assigned vertexes/edges imbalances.

[1] https://investor.fb.com/investor-news/default.aspx.

Our goal is to design a graph partitioning technique that achieves low communication cost and good load balance among computers. The state-of-the art of graph partitioning techniques is *Oblivious* [3] and *HDRF* [6] that are actually implemented in PowerGraph. These techniques generate balanced-size clusters while attempting to reduce communication overhead. However, the communication overhead tends to be high and this degrades the performance, in particular, the number of commuters is large. In contrast, there are other graph clustering techniques [7–9] that are designed to reduce the number of cross-cluster edges. They are expected to reduce the communication overhead; however, the size of the obtained clusters is imbalanced as reported in [8] so we cannot directly apply these techniques to our goal just as they are.

We propose an effective graph partitioning technique that achieves low communication cost and good load balance among computers at the same time. So as to obtain balanced-size clusters, we first generate much more balanced-size clusters than the number of computers by extending the modularity-based clustering, and then merge those clusters into balanced-size clusters by employing the techniques designed for the packing problem [10]. Finally, we convert edge-cut graph into vertex-cut graph, because the modularity clustering is edge-cut-based clustering and most of the recent distributed graph engines are based on vertex-cut graph. We implemented our technique on top of PowerGraph and made evaluations. The results show that our partitioning technique reduces the communication cost so it improves the response time of graph analysis patterns. In particular, it improves the response time of PageRank computation 3.2 times faster at most than HDRF. In addition, we also evaluated how the major graph metrics (the replication factor and load balance factor) correlate with the physical performance measures, the response time, the amount of data transfer between computers, and the imbalance runtime ratio among computers.

The remainder of this paper is organized as follows. Section 2 describes the background of this work. Section 3 describes the detailed design of our technique. Section 4 reports the results of experiments. Section 5 addresses related work, and Sect. 6 concludes this paper.

2 Prelimilary

2.1 Replication Factor and Load Balance Factor

Recent distributed graph processing frameworks (e.g., GraphLab [2] and PowerGraph [3]) have employed vertex-cut method [2, 6] for the graph partitioning since it provides better performance in terms of load balancing among distributed computers. Vertex-cut method is a graph partitioning technique for distributed graph processing; it divides a graph into multiple partitions by replicating cross-cluster vertexes, and it assigns each partition to each computer in the distributed computation environment. In order to qualify the effectiveness of graph partitioning, it is natural choice to use two major metrics called *replication factor* [2] and *load balance factor* [3].

Replication factor [2] is a metric that evaluates communication cost among distributed computers. Replication factor quantifies how many vertexes are replicated over computers compared with the the number of vertexes of the original input graph. The vertex-cut method takes a strategy to replicate cross-cluster vertex and assign the replicas to the computers the adjacent edges of the vertex belong to. In order to keep the consistency of analysis results among distributed computers, we need to communicate and exchange the analysis results among the computers in which the replicated vertexes are located. Thus, we can mitigate the communication cost by keeping the replication factor small. By following the literature [2], we formally define the replication factor RF as follows:

$$\mathrm{RF} = \frac{1}{|V|}\sum_{v \in V} |R(v)|, \tag{1}$$

where V are a set of vertexes, and $R(v)$ is a set of vertexes replicated from vertex v.

Load balance factor is another metric of distributed graph processing that evaluates skewness of loads among distributed computers. Distributed graph processing frameworks using vertex-cut method employ the following equation for evaluating load balance factor:

$$\max_{m \in M} |E(m)| < \lambda \frac{|E|}{|M|}, \tag{2}$$

where E and M are a set of edges and a set of computers, respectively; $E(m)$ is a number of edges that are assigned to computer m. λ is a user-specified parameter that determines the acceptable skewness; user needs to set a value for λ that satisfies $\lambda \in \mathbb{R}$ and $\lambda \geq 1$. That is, Eq. (2) indicates that how large size is acceptable for $E(m)$ compared with the expected number of edges for each computer (i.e., $\frac{|E|}{|M|}$). From Eq. (2), we can conduct the following equation:

$$\lambda = \frac{|M|}{|E|}\max_{m \in M} |E(m)|. \tag{3}$$

In this paper, we call Eq. (3) as *load balance factor*. We employ Eq. (3) for evaluating the load balance efficiency of graph partitioning results.[2]

[2] A partitioning result is well balanced when λ is small.

2.2 Modularity

Our proposed method merges partition pairs for increasing a graph partitioning measure, namely *modularity* [7], so as to reduce the total number of cross-partition edges. In this section, we formally introduce modularity.

Modularity, proposed by Girvan and Newman [7], is widely used to evaluate the quality of graph partitions from global perspective. Modularity is a quality metric of graph partitioning based on *null model*; it measures the difference of the graph structure from the corresponding random graph. Intuitively, graph clustering is to find groups of vertexes that have a lot of inner-group edges and few outer-group edges; optimal partitions are achieved when the modularity is maximized. The modularity Q is formally defined as follows:

$$Q = \sum_{i \in C} \left\{ \frac{|E_{ii}|}{2|E|} - \left(\frac{\sum_{j \in C} |E_{ij}|}{2|E|} \right)^2 \right\}, \tag{4}$$

where C and $|E|$ are a set of partitions and the total number of edges included in graph G, respectively, and E_{ij} is a number of edges between partition i and j.

For finding good partitions, traditional modularity-based algorithms [9, 11, 12] greedily select and merge partition pairs so as to maximize the increase in modularity. However, Eq. (4) is inefficient to evaluate the modularity increase made by merging partition pairs since Eq. (4) needs to compute the complete modularity score for all merging partitions. Instead of computing complete modularity score, existing algorithms (i.e., CNM [11] and Louvain method [12]) conducted an equation of the *modularity gain* $\triangle Q_{ij}$ for efficiently evaluating the modularity increase after merging two partitions i and j as follows:

$$\triangle Q_{ij} = 2 \left\{ \frac{|E_{ij}|}{2|E|} - \left(\frac{\sum_{k \in C} |E_{ik}|}{2|E|} \right) \left(\frac{\sum_{k \in C} |E_{jk}|}{2|E|} \right) \right\}, \tag{5}$$

where $\triangle Q_{ij}$ indicates the modularity gain after merging partition i and j. As we described above, the modularity-based algorithms find a set of partitions that with high modularity Q by iteratively selecting and merging partition pairs that maximize Eq. (5).

In our proposed method, we modify Eq. (5) for finding balanced-size partitions for efficient distributed graph processing; we introduce a new term for balancing the partitioning size [8] into Eq. (5). We present its details in Sect. 3.1.

3 Balanced-Size Clustering Technique

Our goal is to design a graph partitioning technique that achieves low communication cost and good load balance among computers at the same time. We propose an effective graph partitioning technique that achieves low replication factor and good load balance factor. Our technique consists of three phases, balanced-size modularity clustering phase, cluster merge phase, and graph conversion phase as follows.

Balanced-size modularity clustering phase	We first employ a modified modularity proposed by Wakita and Tsurumi [8] that achieves good modularity and mitigates the imbalance of cluster size.
Cluster merge phase	Since modularity clustering generates large number of clusters in general, we need to have additional phase to merge clusters more. Moreover, even if we employ the modified modularity that mitigates imbalanced size of clusters, we still have the imbalance of cluster size. So, we generate much more clusters than the number of computers in the 1st phase, and then merge those clusters into balanced-size clusters until the number of clusters becomes the number of computers by employing techniques designed for graph packing problem.
Graph conversion phase	Finally, we convert edge-cut graph into vertex-cut graph, because the modularity clustering is edge-cut-based clustering and most of the recent distributed graph engines are based on the vertex-cut graph.

3.1 Balanced-Size Modularity Clustering Phase

The goal of this balanced-size modularity clustering phase is to produce fine-grained and well-balanced clusters. In this phase, we iteratively merge cluster pair into clusters so as to increase modularity score while keeping the size of clusters balanced. As we described in Sect. 1, modularity-based clustering algorithms, e.g., CNM [11], generally tend to produce imbalanced sizes of clusters. For mitigating the imbalanced cluster size, we first employ a modified modularity gain $\triangle Q'$, proposed by Wakita and Tsurumi [8], which introduces a heuristic into Eq. (5) for controlling the size of the merged cluster. The modified modularity gain $\triangle Q'_{ij}$ between cluster i and j is defined as follows:

$$\triangle Q'_{ij} = \min\left(\frac{|E_i|}{|E_j|}, \frac{|E_j|}{|E_i|}\right) \triangle Q_{ij}, \tag{6}$$

where E_i is a set of edges included in cluster i. As shown in Eq. (6), we can find clusters that are expected to increase the modularity score since we have $\triangle Q_{ij}$ term in the right-hand side on the equation. In addition, Eq. (6) also evaluates $\min\left(\frac{|E_i|}{|E_j|}, \frac{|E_j|}{|E_i|}\right)$ term, which clearly takes large value when $|E_i|$ and $|E_j|$ are almost same sizes. Hence, the modified modularity gain $\triangle Q'_{ij}$ prefers to merge two clusters whose sizes are similar each other. As a result, Eq. (6) gives large score when two clusters i and j not only contain similar number of inner edges but also show better modularity gain.

For finding fine-grained and well-balanced clusters efficiently, we apply Eq. (6) to the state-of-the-art modularity-based clustering called *incremental aggregation method* [9]. The incremental aggregation method is a modularity-based clustering algorithm that is able to process large-scale graphs with more than a few billion edges within quite short computation time. This is because the method effectively reduces the number of edges to be referenced during the modularity gain computation by incrementally merging cluster pairs. By combining the method and the modified modularity gain shown in Eq. (6), this phase finds the fine-grained and well-balanced clusters efficiently.

In addition, this phase attempts to produce larger number of clusters than user-specified parameter k. The reasons are twofold: (1) Although Eq. (6) is effective in balancing the cluster size, it is not sufficient for the load balance. For further balancing the size of clusters, we additionally perform first-fit algorithm [10] in the next phase, which is an approximation algorithm for the bin packing problem. (2) If we run modularity-based clustering methods until convergence, they automatically determine the number of clusters relying on the input graph topology. In order to control the number of clusters for the distributed machines, this phase needs to run until (a) we can find no cluster pairs that increase the modularity score, or (b) the number of clusters produced in this phase reaches $a \times k$ where $a \in \mathbb{R}$ is a user-specified parameter such that $a > 1$.

3.2 Cluster Merge Phase

Algorithm 1 Cluster merge

Input: $\mathbb{C}, k$
Output: $\mathbb{R}$

$\mathbb{R} \leftarrow top\_k\_clusters(\mathbb{C}, k)$
$\mathbb{C} \leftarrow \mathbb{C} - \mathbb{R}$
while $\mathbb{C} \neq \emptyset$ **do**
  $m \leftarrow \arg\min_{r \in \mathbb{R}} \{|inner\_edges(r)|\}$
  $\mathbb{N} \leftarrow neighbors(m) \cap \mathbb{C}$
  if $\mathbb{N} = \emptyset$ **then**
    break
  end if
  for each $n \in \mathbb{N}$ **do**
    merge cluster m and n, generate cluster m'
    delete n from $\mathbb{C}$
    delete m from $\mathbb{R}$, insert m' into $\mathbb{R}$
    if $m' \neq \arg\min_{r \in \mathbb{R}} \{|inner\_edges(r)|\}$ **then**
      break
    end if
  end for
end while
$merge\_flag = true$
while $\mathbb{C} \neq \emptyset$ **and** $merge\_flag$ **do**
  $merge\_flag = false$
  for each $c \in \mathbb{C}$ **do**
    $\mathbb{N} \leftarrow neighbors(c) \cap \mathbb{R}$
    if $\mathbb{N} \neq \emptyset$ **then**
      $merge\_flag = true$
      $m \leftarrow \arg\min_{n \in \mathbb{N}} \{|inner\_edges(n)| + |cut\_edges(n, c)|\}$
      merge cluster m and c, generate cluster m'
      delete c from $\mathbb{C}$
      delete m from $\mathbb{R}$, insert m' into $\mathbb{R}$
    end if
  end for
end while
for each $c \in \mathbb{C}$ **do**
  delete c from $\mathbb{C}$
  $\mathbb{N} \leftarrow neighbors(c) \cap \mathbb{C}$
  while $\mathbb{N} \neq \emptyset$ **do**
    for each $n \in \mathbb{N}$ **do**
      merge cluster c and n, generate cluster c'
      delete n from $\mathbb{C}$
      $c \leftarrow c'$
    end for
    $\mathbb{N} \leftarrow neighbors(c) \cap \mathbb{C}$
  end while
  $m \leftarrow \arg\min_{r \in \mathbb{R}} \{|inner\_edges(r)|\}$
  merge cluster m and c, generate cluster m'
  delete m from $\mathbb{R}$, insert m' into $\mathbb{R}$
end for
return $\mathbb{R}$

The idea of producing balanced-size clusters is to employ the techniques developed for the packing

problem [10]. That is, given various size of items, we pack them into fixed number of containers with the same size. Since we generated more clusters than the number of computers at the last phase, we pack those clusters into balanced-size containers by performing first-fit algorithm. In addition, we choose an adjacent cluster of a given cluster and pack them into the same container during first-fit algorithm, so that we can keep the number of cross-cluster edges small.

The detail is as follows. Given we have many clusters produced at the balanced-size modularity clustering phase, we choose k (number of computers) largest clusters as seed clusters and put them into different containers. Then, we repeatedly merge the smallest seed cluster with its adjacent cluster until there is no adjacent cluster to seed clusters. After that, there may be clusters that are not connected to any seed clusters, that is, the clusters are isolated from any seed clusters. We pick up a cluster from the isolated ones, merge reachable clusters from it, and put the merged cluster into the container with the smallest number of inner edges.

The pseudocode of this phase is shown in Algorithm 1. The symbols and their definitions used in the code are summarized in Table 1. The input is clusters $\mathbb{C}$, and the specified number of output clusters is k. Clusters $\mathbb{C}$ are obtained at the balanced-size modularity clustering phase. First, we choose k clusters that have the largest number of inner edges from input clusters $\mathbb{C}$. We treat them as seed clusters and put them into output clusters $\mathbb{R}$ (line 1). In the following procedure, we pick up other clusters from $\mathbb{C}$ and merge them with the seed clusters until no cluster is left in $\mathbb{C}$. The procedure consists of three steps. In the first step, so as to balance the size of the seed clusters while keeping the number of cross-cluster edges small, we choose the smallest seed cluster, pick up its adjacent cluster in $\mathbb{C}$, and merge the seed cluster with the adjacent cluster (line 3–17). We repeat this merge process until there is no adjacent cluster to the smallest seed clusters left in $\mathbb{C}$. In the second step, we pick up a cluster in $\mathbb{C}$ and merge it with its adjacent and the smallest seed cluster (line 18–33). We repeat this merge process until there is no adjacent cluster to the seed clusters left in $\mathbb{C}$. Now, there may be clusters in $\mathbb{C}$ that are not connected to any seed clusters. In the final step, we treat the seed clusters in $\mathbb{R}$ as containers of the packing problem. We pick up a cluster in $\mathbb{C}$ (line 32–33), merge it with its reachable clusters in $\mathbb{C}$ (line 34–42), and put it to the smallest seed cluster (container) (line 43–45).

Table 1 Definitions of symbols used in Algorithm 1

Symbol	Definition
$\mathbb{C}$	Input cluster set
k	Specified number of output clusters
$\mathbb{R}$	Output cluster set
$top\_k\_clusters(\mathbb{C}, k)$	Top-k clusters $\in \mathbb{C}$
$inner\_edges(c)$	Inner edges of cluster c
$neighbors(c)$	Adjacent clusters of cluster c
$cut\_edges(n, m)$	Cut edges between cluster n and m

Example 1 Figure 1 depicts an example of the cluster merge phase, the initial state is on the left, and the final state is on the right. Each circle represents cluster, and the number located at the center of the circle shows the number of inner edges in the cluster. The number assigned to an edge shows the number of the cross-cluster edges. The dotted shape represents seed cluster (container). (1) In the initial state, two largest clusters (cluster 1 and cluster 2) are chosen as seed clusters. (2) The smallest seed cluster (cluster 2) and its one of adjacent clusters (cluster 3) are merged. (3) Still the merged seed cluster (containing cluster 2 and cluster 3) is the smallest seed cluster [the size is 35 (20 + 5 + 10)], so we continue to merge it with its adjacent cluster, cluster 5. (4) Now the merged seed cluster size is 55, the smallest cluster changes to cluster 1. Then, cluster 1 is merged with its adjacent cluster, cluster 4, and the size becomes 65. (5) Now, there is no adjacent cluster to any seed clusters, so we put the isolated cluster, cluster 6, into the smallest seed cluster, cluster 2, as shown in the final state in Fig. 1.

3.3 Graph Conversion Phase

So far, we have obtained k clusters of edge-cut graph. In this final phase, we convert edge-cut graph into vertex-cut graph, since most of the recent distributed graph engines are based on the vertex-cut graph. This design is based on the fact that vertex-cut graph is more efficiently balanced than edge-cut graph [3, 13]. To convert edge-cut graph to vertex-cut graph, we have to convert cross-cluster edge to cross-cluster vertex by choosing either two sides of cross-cluster edge as cross-cluster vertex. Let u is chosen as cross-cluster vertex and v is not for cross-cluster edge $e(u, v)$. The cross-cluster edge $e(u, v)$ is assigned to the cluster to which non-cross-cluster vertex v belong. We choose cross-cluster vertexes so that the size of the clusters to be balanced. This procedure is simple but affects largely the load balance.

4 Experiments

We implemented our proposal, balanced-size clustering technique, on top of one of the recent distributed graph processing frameworks, PowerGraph [3]. We made following experiments to validate the effectiveness of our graph partitioning technique.

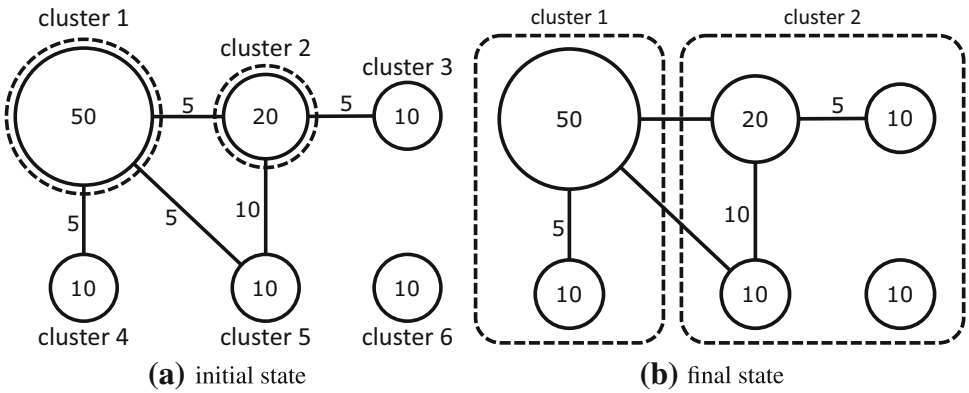

Fig. 1 Example in cluster merge phase. The initial state is on the *left*, and the final state is on the *right*. *Each circle* represents cluster, and the number located at the center of the circle shows the number of inner edges in the cluster. The number assigned to an edge shows the number of the cross-cluster edges. The *dotted shape* represents seed cluster (container) **a** initial state, **b** final state

Partitioned graph quality	We evaluated the effectiveness of partitioned graph by using the major metrics, replication factor [Eq. (1)] and load balance factor [Eq. (2)].
Performance for graph analysis	We evaluated the runtime, the amount of data transfer between computers, and the imbalance runtime ratio among computers during graph analysis. In addition, we also evaluated how the major graph metrics, the replication factor and load balance factor, correlate with the physical performance measures, the response time, the amount of data transfer between computers, and the imbalance runtime ratio among computers.
Scalability	We evaluated the response time of graph analysis, graph partitioning time, and the sum of both by varying the number of computers.

We compared our graph partitioning technique to other techniques, a random partition, *Oblivious* [3], and *HDRF* [6]. The random partitioning is a naive approach that randomly assigns vertexes/edges to distributed computers. The Oblivious is a heuristic technique that balances the size of partitions and reduces the replication factor. The HDRF is a technique improved from Oblivious and actually provides better graph partitions than Oblivious does for various graphs. We used two variations of our graph partitioning technique in the 1st phase; the original modularity clustering and the balanced-size modularity clustering. They are denoted as *modularity* and *balanced-size* in figures, respectively. For the parameter setting, we choose the number of clusters the 1st phase generates according to the graph size; we set more clusters to generate as input graph size increases.

4.1 Benchmark

We used real graph data shown in Table 2 and three typical graph analysis patterns as follows.

1. PageRank [14]: one of the link-based ranking techniques designed for web pages.
2. SSSP (single-source shortest path): computing the shortest paths to all vertexes from a given vertex.
3. CC (connected component): detecting sub-graphs (components) connected with edges.

4.2 Setting

The experiments were made on Amazon EC2, r3.2xlarge Linux instances. Each instance has CPU Intel(R) Xeon(R) CPU E5-2670 v2, 2.50 GHz (four cores) with 64 GB RAM. The network performance between instances was 1.03 Gbps. The hard disks delivered 103 MB/s for buffered reads. We used g++4.8.1 with –O3 optimization for PowerGraph and all partitioning techniques. We chose synchronous engine of PowerGraph to ensure the preciseness of the analysis results.

4.3 Partitioned Graph Quality

We evaluated the effectiveness of partitioned graph by using the major metrics, replication factor [Eq. (1)] and load balance factor [Eq. (2)] for the graph data in Table 2.

Table 2 Real-world graph data

Dataset	Short name	$\|V\|$	$\|E\|$	Modularity
email-EuAll [15]	Eu	265,214	420,045	0.779
web-Stanford [15]	St	281,903	2,312,497	0.914
com-DBLP [15]	DB	317,080	1,049,866	0.806
web-NotreDame [15]	No	325,729	1,497,134	0.931
amazon0505 [15]	am	410,236	3,356,824	0.852
web-BerkStan [15]	Be	685,230	7,600,595	0.930
web-Google [15]	Go	875,713	5,105,039	0.974
soc-Pokec [15]	Po	1,632,803	30,622,564	0.633
roadNet-CA [15]	CA	1,965,206	2,766,607	0.992
wiki-Talk [15]	Ta	2,394,385	5,021,410	0.566
soc-LiveJournal1 [15]	Li	4,847,571	68,993,773	0.721
uk-2002 [16]	uk	18,520,486	298,113,762	0.986
webbase-2001 [16]	ba	118,142,155	1,019,903,190	0.976

4.3.1 *Relationship Between Modularity and Replication Factor*

Our technique is based on modularity clustering so as to decrease the number of cross-cluster edges. We investigated the relationship between the modularity value[3] of the real graph data and how our technique improves replication factor for those data compared with random partitioning and HDRF. In Fig. 2, *X*-axis shows the modularity value and *Y*-axis shows the replication factor ratio of the partitions obtained by our technique to those obtained by random partitioning and HDRF. As expected, we observe that our technique provides better replication factors than other techniques and that the replication factor is improved more as the modularity value of the graph increases.

4.3.2 *Replication Factor*

Figure 3 shows the results of the experiments for replication factor by varying the number of computers, 8, 16, 32, 48, 64. The figure includes only the three largest graph data, soc-LiveJournal1, uk-2002, webbase-2001. We omit others here because they are similar results to the above three graph data. We set the number of clusters the 1st phase generates at 4000, 8000, 160,000 for soc-LiveJournal1, uk-2002, webbase-2001, respectively.

We observe that our technique achieves the best among others and the advantage increases as the number of computers increases. Only for soc-LiveJournal1, the variation that uses the original modularity clustering in the 1st phase performs better than the variation that uses the balanced-size modularity clustering. We guess this is caused by the fact that the modularity of soc-LiveJournal1 (0.721) is relatively lower than those (0.986 and 0.976) of uk-2002 and webbase-2001 (see Fig. 2), so the balanced-size modularity clustering could not improve the replication factor as the original modularity clustering.

4.3.3 *Load Balance Factor*

Figure 4 shows the results of the experiments for load balance factor. We observe that the variation that uses the original modularity clustering seriously inferior to others. This is because the primary goal of the Oblivious and HDRF is to generate balanced-size clusters and decreasing the replication factor is secondary. We also observe that the balanced-size modularity clustering effectively mitigates the load balance factor to the original modularity clustering.

4.4 Performance for Graph Analysis

We evaluated the runtime time, the amount of data transfer between computers, and the imbalance runtime ratio among computers during graph analysis executed on PowerGraph. We fixed the number of computers at 64.

4.4.1 *Runtime*

Figure 5 shows the runtime results for analysis patterns. The runtime results in *Y*-axis are normalized to random partitioning result. As we can see in the figure, our technique performs best among others. In general, our technique is more effective as the modularity of graph increases (soc-LiveJournal1 0.721 → webbase-2001 0.976 → uk-2002 0.986), mainly because the amount of data transfer is reduced more for the graph with larger modularity (we will see in Fig. 6). Also the response time is not correlated so much with the imbalance runtime ratio (we will see in Fig. 7). For the largest modularity case of uk-2002, our

[3] We set the number of partitions at 64.

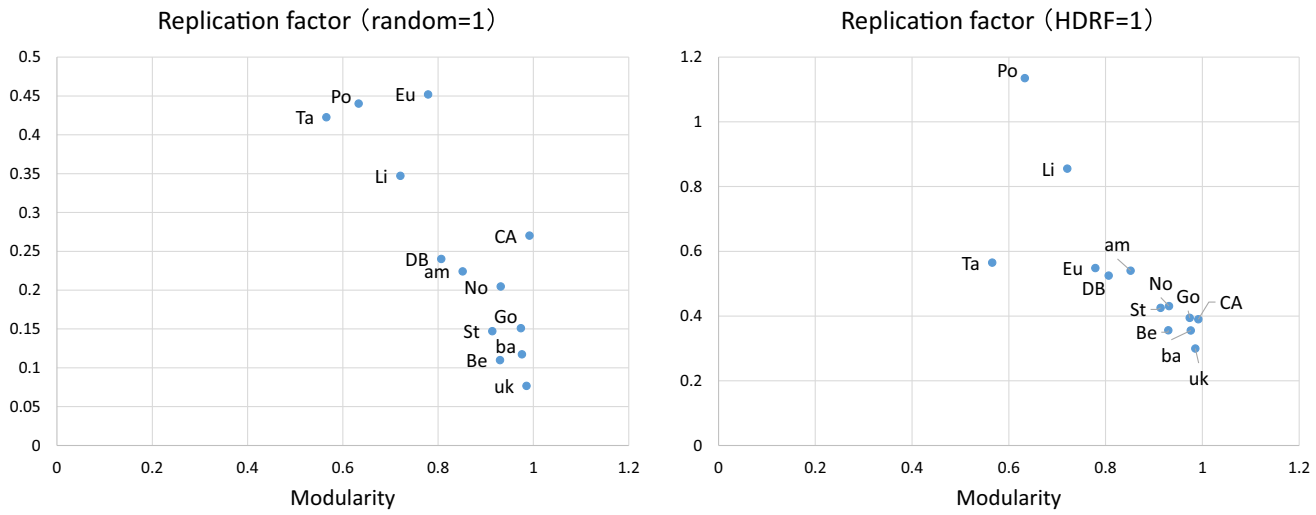

Fig. 2 Relationship between Modularity and replication factor for graph data in Table 2. *Y*-axis shows the replication factor ratio of the partitions obtained by our technique to those obtained by random partitioning and HDRF

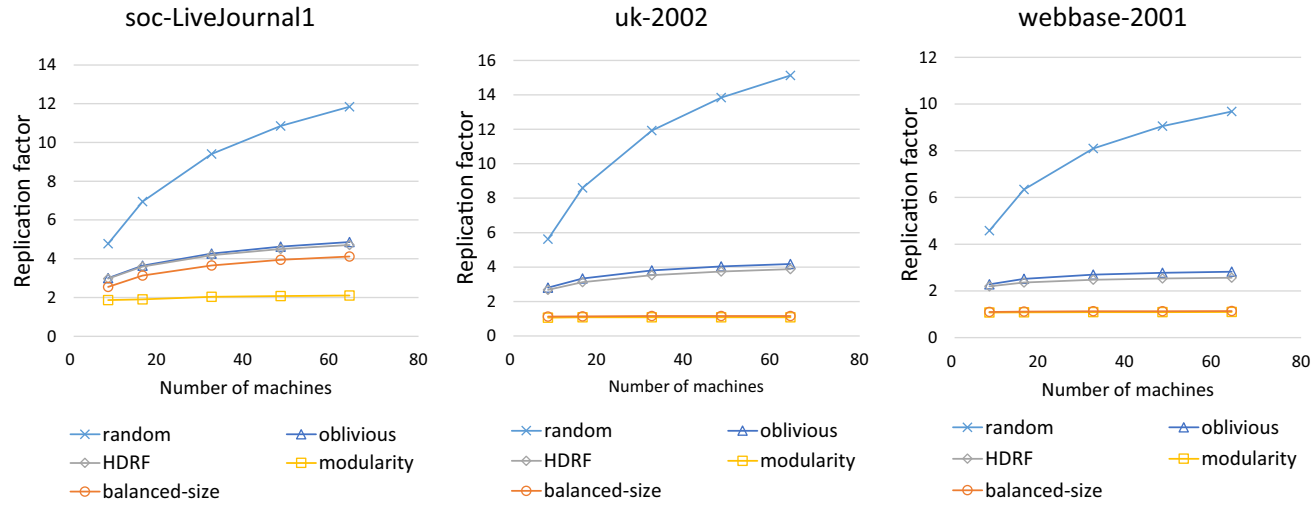

Fig. 3 Scalability experiments for replication factor

technique is 3.2, 1.2, 2.2 times faster in PageRank, SSSP, CCC, respectively, than HDRF. For soc-LiveJournal1, where the modularity is the smallest among others, the variation that uses the balanced-size modularity clustering in the 1st phase provides higher performance than the one that uses the original modularity clustering.

4.4.2 Amount of Data Transfer

Figure 6 shows the average amount of data transfer between computers for analysis patterns. The results in *Y*-axis are normalized to random partitioning result. By comparing this figure with the replication factor experiments in Fig. 3, the amount of data transfer is highly correlated with the replication factor. For the largest modularity case of uk-2002, our technique most effectively reduces the amount of data transfer by 94%, 62%, 95% of HDRF in PageRank, SSSP, CCC, respectively. We guess the runtime improvement achieved by our technique is caused by not only the reduction ratio of data transfer but also its actual amount of data transfer. Our technique improves the runtime of PageRank most, because both the reduction rate of data transfer and the actual amount of data transfer are large. The actual amount of data transfer is 21 GB in PageRank, 0.17 GB in SSSP, 1.5 GB in CC for the case of uk-2002 in random partitioning.

4.4.3 Imbalance Runtime Ratio

Figure 7 shows the imbalance runtime ratio for analysis patterns. The runtime indicates CPU time and excludes network IO wait. The imbalance runtime ratio is defined as

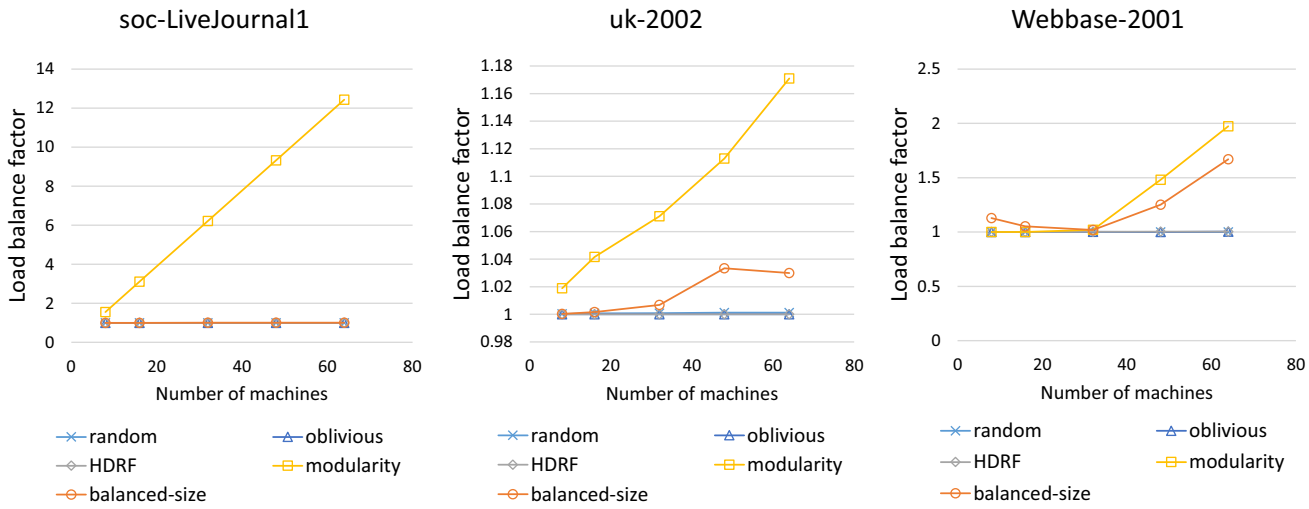

Fig. 4 Scalability experiments for load balance factor

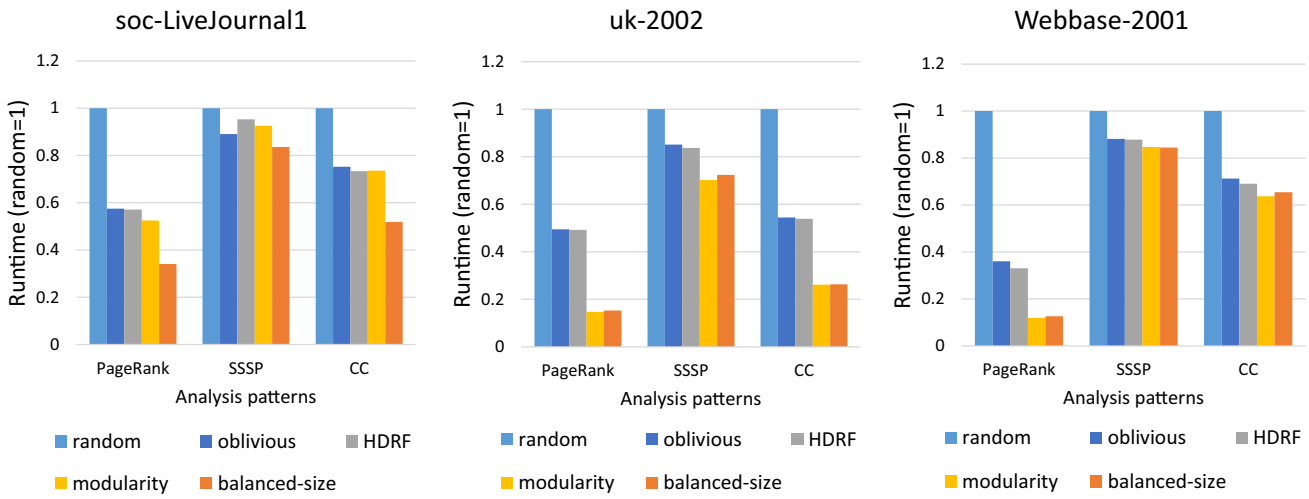

Fig. 5 Runtime experiments for analysis patterns (*Y*-axis is normalized to random partitioning result)

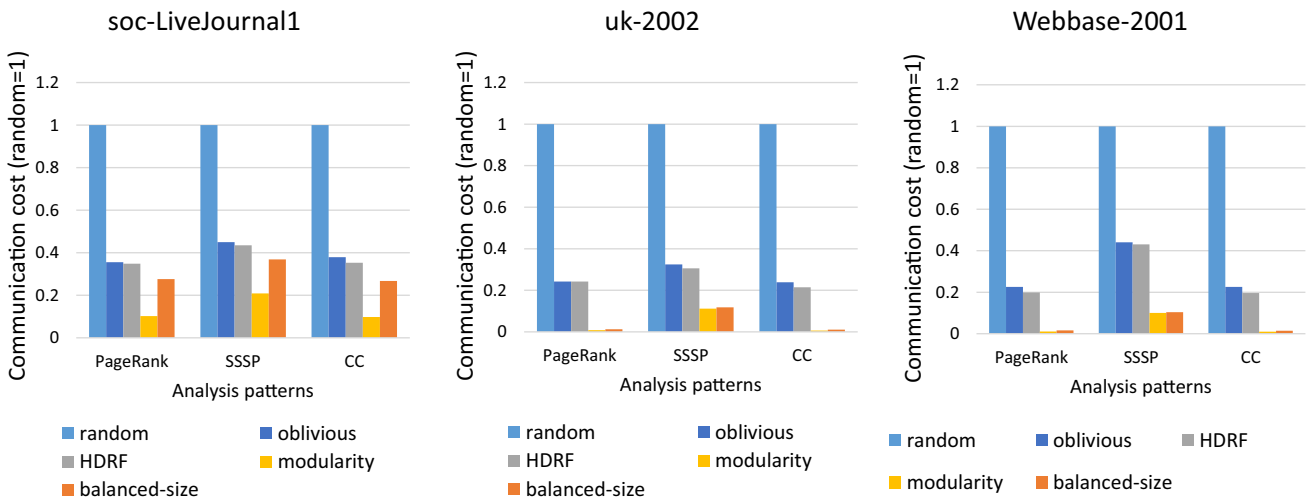

Fig. 6 Average amount of data transfer between computers for analysis patterns (*Y*-axis is normalized to random partitioning result)

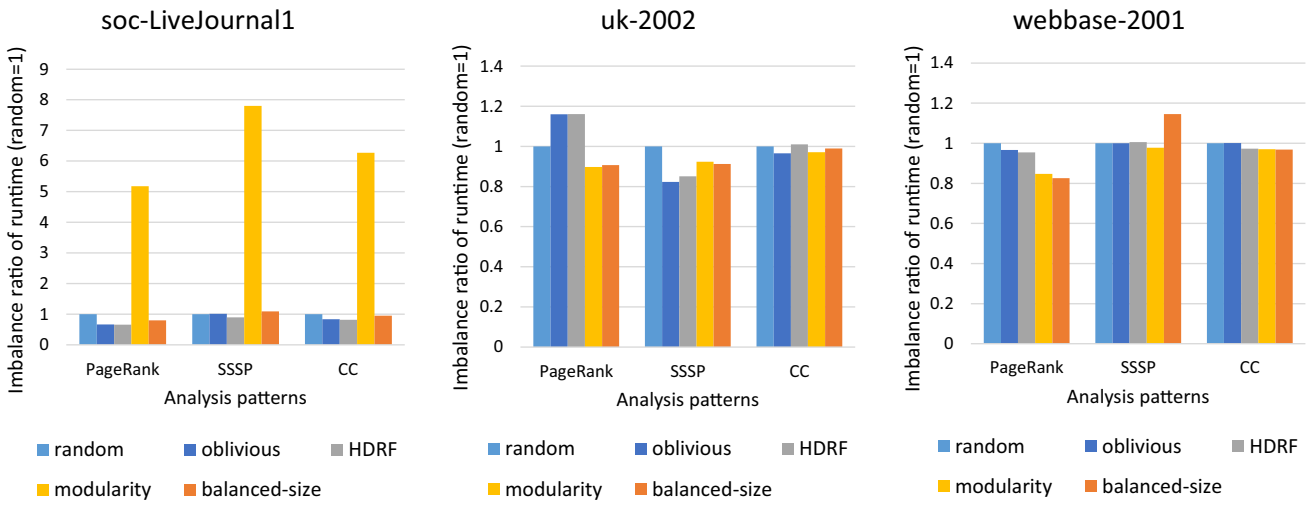

Fig. 7 Imbalance runtime ratio for analysis patterns (*Y*-axis is normalized to random partitioning result)

the ratio of the slowest computer's runtime to the average runtime. Again, the results in *Y*-axis are normalized to random partitioning result. By comparing this figure with the load balance factor experiments in Fig. 4, the imbalance runtime ratio is correlated with the load balance factor. In particular for the case of the variation that uses the original modularity clustering in the 1st phase for soc-LiveJournal1. Except that, the results are comparable to all techniques for all graph data. Notice that, for our technique that uses the balanced-size modularity clustering in the 1st phase, even if its load balance factor is inferior to others (see in Fig. 4), the imbalance runtime is comparable to others. In addition, the imbalance ratio changes to analysis patterns. So, we conjecture that there should be other factors than load balance factor that affect the imbalance runtime depending on analysis patterns.

4.5 Scalability

We evaluated the response time of graph analysis, graph partitioning (ingress) time, and the sum of both (total time) by varying the number of computers. Figure 8 shows runtime results for the largest graph data, webbase-2001, and PageRank analysis pattern. The analysis results show that our technique scales well to the number of computers and achieves best among others. For the graph ingress time, random partitioning is fastest because it chooses a computer to assign a new edge randomly. Our technique is scalable since we extend to use the state-of-the-art modularity-based clustering [9] in the 1st phase and the cost of the 2nd and 3rd phases does not depend on the number of computers. Notice that both Oblivious and HDRF are not scalable. The ingress time of Oblivious and HDRF gets worse to the number of computers. We investigated the implementation of Oblivious and HDRF and found that they made a linear search on the computer list to determine which computer stores the smallest number of assigned edges. For the total time, our technique is the best, in particular, the variation that uses the balanced-size modularity clustering in the 1st phase.

5 Related Work

In the line of the work for efficient distributed graph processing, the problem of finding better graph partitions has been studied in recent decades. A recent survey paper on vertex-centric frameworks summarizes various types of graph partitioning techniques [17]. The major approach is twofold: *edge-cut method* and *vertex-cut method*.

Edge-cut method The edge-cut method is a graph partitioning approach that divides a graph into sets of sub-graphs by cutting edges so as to reduce the number of cross-partition edges. In the distributed graph processing, the edge-cut method assigns each sub-graph to each computer. METIS, proposed by Karypis and Kumar in 1998 [18], is one of the representative partitioning algorithms that focuses on reducing the number of cross-partition edges via the edge-cut method. The problem of edge-cut method is that it cannot avoid load imbalance for typical graphs that follow the power law distribution [19]. We explain the detail more in the vertex-cut method part.

Vertex-cut method The vertex-cut method is another type of partitioning technique that attempts to reduce the number of cross-partition vertexes. As we described above, the edge-cut method splits a graph into sets of sub-graphs by cutting edges. In contrast, the vertex-cut method divides a graph by splitting vertexes. Most of the recent distributed graph engines use vertex-cut methods, because vertex-cut graph is more efficiently balanced than edge-cut

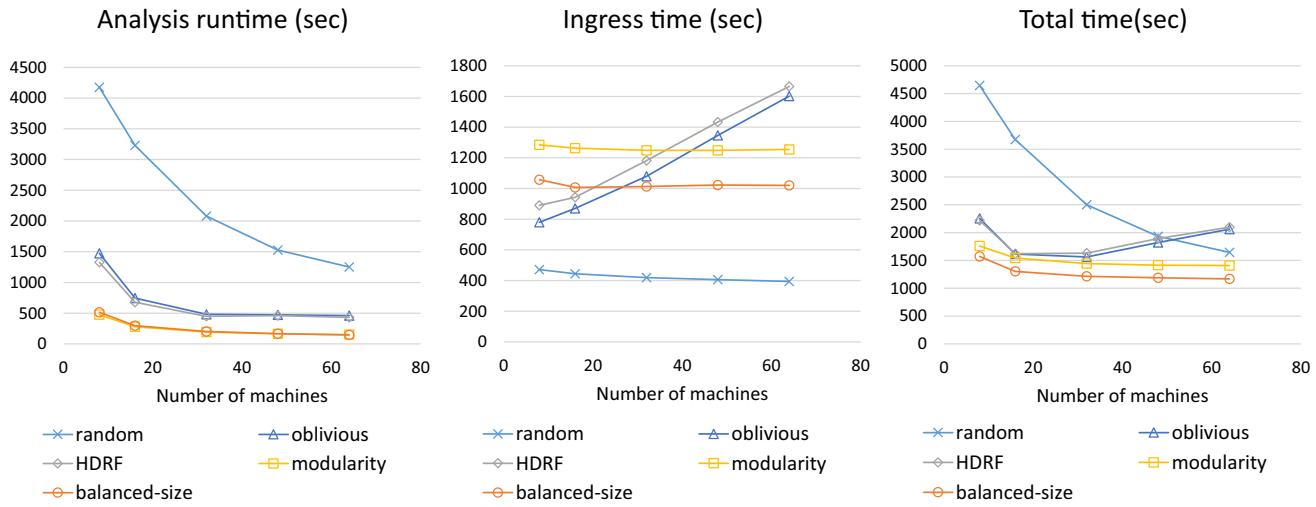

Fig. 8 Scalability experiments for analysis runtime, graph ingress time, and total time (for webbase-2001 dataset and PageRank analysis pattern)

graph [3, 13]. Typically, graph usually follows the power law distribution so it tends to include super-vertexes, that is, the number of their connected edges is tremendously large. Those super-vertexes affect largely load imbalance, so the idea of the vertex-cut method is to reduce the load imbalance by splitting the super-vertexes. In the family of the vertex-cut methods, Oblivious [2] and HDRF (High-Degree (are) Replicated First) [6] are the state-of-the-art algorithms. These algorithms are stream-based algorithms: Every edge is read from input file, and it is immediately assigned to a computer; and thus, they are scalable to large-scale graphs and achieve better load balance performance. Specifically, Oblivious assigns an incoming edge to a computer, so that it can reduce the number of cross-vertexes spanned among computers. HDRF divides edges into partitions by splitting high-degree vertexes in order to reduce the total number of cross-vertexes.

6 Conclusion

We proposed a graph partitioning technique that efficiently partitions graphs with good quality so that it achieves high performance for graph analysis by reducing the communication cost and by keeping good load balance among computers. We extend modularity-based clustering and integrate it with the techniques for the graph packing problem. We implemented our technique on top of distributed graph engine, PowerGraph, and made intensive experiments. The results show that our partitioning technique reduces the communication cost so it improves the response time of graph analysis patterns. In particular, PageRank computation is 3.2 times faster at most than HDRF, the state-of-the art of streaming-based partitioning approach. In addition, we observed that the replication factor and load balance factor correlate with the amount of data transfer and the imbalance runtime ratio, respectively, and that the response time is correlated with the replication factor but not with the load balance factor so much.

Possible future work is as follows. (1) There is a trade-off between the communication cost and load balance depending on the number of computers. We optimize the trade-off problem by fixing the number of computers in this paper, but one future work is to optimize the number of computers depending on the input graph and analysis patterns. (2) There is a still room improving more on the replication factor and load imbalance and achieving efficient graph clustering.

References

1. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of SIGMOD
2. Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein JM (2012) Distributed GraphLab: a framework for machine learning and data mining in the cloud. PVLDB, 5(8):716–727
3. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of OSDI
4. Xin RS, Gonzalez JE, Franklin MJ, Stoica I (2013) GraphX: a resilient distributed graph system on Spark. In: Proceeding of GRADES
5. Suri S, Vassilvitskii S (2011) Counting triangles and the curse of the last reducer. In: Proceedings of WWW

6. Petroni F, Querzoni Leonardo, Daudjee K, Kamali S, Iacoboni G (2015) HDRF: stream-based partitioning for power-law graphs. In: Proceeding of CIKM
7. Newman MEJ, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69, 026113
8. Wakita K, Tsurumi T (2007) Finding community structure in mega-scale social networks. In: Proceedings of WWW
9. Shiokawa H, Fujiwara Y (2013) Fast algorithm for modularity-based graph clustering. In: Proceeding of AAAI, Onizuka
10. Dósa G, Sgall J (2013) First fit bin packing: a tight analysis. In: Proceeding of STACS
11. Clauset A, Newman MEJ, Moore C (2004) Finding community structure in very large networks. Phys Rev E 70:066111
12. Blondel VD, Guillaume J, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. J Stat Mech Theory Exp. doi:10.1088/1742-5468/2008/10/P10008
13. Bourse F, Lelarge M, Vojnovic M (2014) Balanced graph edge partition. In: Proceeding of KDD
14. Page L, Brin S, Motwani R, Winograd T (1999) The PageRank citation ranking: bringing order to the web. Technical report
15. Stanford Large Network Dataset Collection (2014) http://snap.stanford.edu/data/. Accessed 31 Jan 2017
16. Laboratory for Web Algorithmics (2002) http://law.di.unimi.it. Accessed 31 Jan 2017
17. McCune RR, Weninger T, Madey G (2015) Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput Surv 48(2):25
18. Karypis G, Kumar V (1999) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
19. Faloutsos M, Faloutsos P, Faloutsos C (1999) On power-law relationships of the internet topology. In: Proceeding of SIGCOMM