7 Network Performance Aware Graph Partitioning for Large Graph Processing Systems in the Cloud

Rishan Chen, Xuetian Weng, Bingsheng He, Byron Choi, and Mao Yang

CONTENTS

7.1 INTRODUCTION

A wide variety of recent applications model their data in graphs/networks such as social networks, web graphs, and protein–protein interaction networks. Efficient processing for large graph data poses new challenges for almost all components of state-of-the-art data management systems. To list a few examples: (i) graph data are complex structures and cannot be efficiently stored as relational tables; (ii) the access patterns of large graph processing are complex, which results in inefficient disk accesses or network communications; and (iii) last but not least, to tackle scalability issues, graph processing must be efficiently distributed in a networked environment.

Researchers have been actively proposing many innovative solutions to address the new challenges of large graph processing. In particular, a notable number of techniques have recently been proposed to utilize the cloud. The objectives of this chapter are (i) to introduce typical examples of large graph processing, (ii) to give an overview of existing cloud-based graph processing platforms, and more importantly, (iii) to emphasize a network performance aware data partitioning approach, which bridges large graph processing and cloud-based platforms. In particular, the network bandwidth may not be uniform across the large network in a cloud; a network with higher bandwidth between its machines can support more intermachine computation.

The chapter is structured as follows. We survey some typical examples of large graph processing in Section 7.2. In Section 7.3, we list some representative cloud-based graph processing platforms. Section 7.4 presents the network unevenness in cloud-based systems and Section 7.5 introduces network performance aware graph partitioning. For the

completeness of discussions, Section 7.7 gives an introduction to existing graph partitioning approaches, although they may not be related to the cloud technologies. A discussion of open problems is provided in Section 7.8. We summarize the chapter in Section 7.9.

7.2 APPLICATIONS OF LARGE GRAPHS

Large graphs have arisen in a wide range of data-intensive applications. To begin our discussions, we first describe a small and non-exhaustive set of typical examples of large graphs and their applications.

7.2.1 Social Networks

In social networks, nodes often represent users and edges may often represent relationships between users (friendships). Today, there are plenty of large social networks. For example, the social network of Facebook consisted of 1 billion nodes and more than 100 billion edges in 2012 [70]. The largest publicly available social network (contributed by Yahoo!*) consists of more than 1 billion nodes. The social network of LinkedIn contained almost 218 million nodes in the first quarter of 2013 [54]. The project FlockDB manages social graphs with more than 13 billion edges [40]. Moreover, social networks are evolving at an unprecedented rate. For example, it has been reported that, between 2004 and 2012, the Facebook network increased from roughly 1 million to 1 billion users [70].

Analysis on social networks has become a hot research topic. Work has been conducted identifying and searching user communities from the networks, and studies have been carried out to estimate the diameter and the radius of a network (e.g., [42]). These studies show how users are connected and indicate which users are outliers of the network. It is reported that the small-world phenomenon has been found in social networks [42]. In practice, despite the large number of users on social networks, it is often a user's close friends who often have the most influence on him/her. It is desirable to determine two- or three-hop friend lists for a social network user. Another application of the networks is to help organizing activities. An organizer can find not only a group of his/her close friends, but also groups that contain people who are close friends of each other.

7.2.2 Web Graphs

Another example of large graphs is the WWW graph. The nodes represent web pages and edges represent hyperlinks. Google estimates that there are over 1 trillion web pages. The indexed web contained at least 4.6 billion web pages as of June 2013.† Today, the WWW graphs for experimentation contain more than 20 billion web pages and 160 billion hyperlinks. The web page hyperlink connectivity graph of Yahoo! AltaVista of 2002 is publicly available.‡ The well-known application of

* Webscope from Yahoo! Labs. Graph, and Social Data. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

† WorldWideWebSize.com: http://www.worldwidewebsize.com/.

‡ Webscope from Yahoo! Labs. Graph and Social Data. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

the WWW graph is the computation of web pages' PageRank [63] for web searches. Let the engineering details such as damping factor alone, the PageRank algorithm iteratively computes the PageRank of each page from the PageRanks of the pages that link to it. The algorithm terminates when the PageRanks of the pages converge. The PageRank algorithm is often used as an example to illustrate performance characteristics of cloud-based platforms.

7.2.3 Information Networks

Resource Description Framework (RDF) has been an official W3C recommendation for the semantic web. The triplets of RDF naturally form a graph. Among others, RDF has been applied to knowledge bases, such as DBpedia [6]. The ontology of DBpedia derived from Wikipedia contains 3.7 millions of "things" and 400 millions of facts.* Such data are particularly useful for users to formulate complex queries about the information represented in the RDF. Applications of the semantic web continue to emerge each year [1].

Search engine providers are actively engaged in introducing semantics for next generation search engines (e.g., Probase [78]).† A recent report of the graph-based knowledge base Satori [13] from Microsoft, which enhances the search capabilities of Bing, consists of more than 300 million nodes and 800 million edges. Google's knowledge graph has 570 million objects and 18 billion facts about the relationships between different objects. The knowledge graphs are expected to enhance the ranking mechanisms of search results.

7.2.4 Miscellaneous

Other examples of large graphs are the citation relationship of research articles, relationships between US patents,‡ Wordnet,§ communication networks, transportation or road networks, and many others. Some of these graphs can be found in the a nice collection of graphs of the Stanford Network Analysis Project (SNAP) [53].

7.3 CLOUD-BASED GRAPH PROCESSING PLATFORMS

As described in the previous section, graph data are ubiquitous and their volume is ever increasing. New computationally and data-intensive analysis tasks on graphs are continuously being reported. The deployments of applications on such data have been moving from a small number of high-performance servers or super computers [31,46] toward a *cloud* with a large number of commodity servers [43,58].

A number of general-purpose development platforms such as MapReduce [23], its open-source variant, Hadoop [33], and Dryad [37] have been proposed to help users to develop custom applications on the cloud, without worrying about the complexity beneath the cloud. For instance, data may be stored in distributed and replicated file

* DBpedia SPARQL Benchmark: http://aksw.org/Projects/DBPSB.html.
† Probase: http://research.microsoft.com/probase/.
‡ US Patent: http://vlado.fmf.uni-lj.si/pub/networks/data/patents/Patents.htm.
§ Wordnet: http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.zip.

systems such as GFS [30] or BigTable [17]. Such systems are suitable for processing flat data structures, not just graph structured data. In particular, it is known that much graph analysis inherently involves random access and direct adoption of the technologies for flat files or relations may lead to high (network) communication costs in the cloud. It is desirable to have a graph-processing platform that automatically handles optimization details for users.

Graph processing platforms for the cloud have recently been proposed. Most of these platforms (e.g., [42,43,81]) are built on top of MapReduce [23]. In this section, we give a brief survey of some representative solutions.

7.3.1 Survey of Existing Systems

7.3.1.1 Pregel

Pregel [58] is a vertex-oriented graph processing engine that implements a Bulk Synchronous Parallel (BSP) model. Pregel passes computational results between workers. It provides a user-defined API *Compute*() executed on vertices. In one iteration of BSP (i.e., *superstep* in Pregel's terminology), Pregel executes *Compute*() on all the vertices in parallel. Messages are passed over the network. Vertices vote to halt if they have no work to do.

7.3.1.2 PEGASUS

PEGASUS [43] is an open-source Hadoop-based library that supports typical graph mining operations including PageRank, spectral clustering, diameter and radius estimations, and connected components. An important observation is that many such mining operations can be readily expressed as an iterative matrix-vector multiplication. PEGASUS therefore proposes a scalable, highly optimized primitive called generalized iterated matrix–vector multiplication that includes block multiplication, clustered edges, and diagonal block iteration.

7.3.1.3 HADI

HADI [42] is a graph mining implementation (developed on Hadoop) that estimates the radii and diameter of a large graph. To tackle the scale of large graphs, HADI proposes an approximation algorithm implemented and optimized for the cloud framework Hadoop/MapReduce.

7.3.1.4 Surfer

Surfer [18] is a large graph processing engine that provides two primitives for developing applications on the cloud: MapReduce and propagation. MapReduce is useful for applications processing flat data structures. In comparison, the second primitive propagation operation is designed for developing edge-oriented tasks on large graphs. A prototype [19] is developed on top of Pregel extended with a network performance-aware partitioning framework.

7.3.1.5 Trinity

Trinity [67] is a distributed memory-based general purpose graph engine. An observation is that MapReduce implementation of graph processing can lead to huge I/O

and communication overhead. Trinity exploits the memory of the machines in the cloud forming a "memory cloud," which enables fast random data access, which is particularly useful for computation on graphs. In addition, Trinity consists of a native graph storage engine. These techniques significantly speed up large graph processing. Trinity supports both transactional and batched graph processing.

7.3.1.6 GraphLab

GraphLab [56] is specially designed for machine learning and data mining algorithms, which are not naturally supported by MapReduce. The GraphLab abstraction enables developers to specify asynchronous, dynamic, graph-parallel computation while ensuring data consistency and achieving a high degree of parallel performance in the shared-memory setting. GraphLab uses an asynchronous parallel model different from the BSP model used by Pregel. Additionally, The GraphLab framework has been extended to the distributed setting while preserving strong data consistency guarantees [55].

Other cloud-based solutions for graph processing include the following. DisG [81] is an ongoing project for web graph reconstruction using Hadoop. Pujol et al. [65] studied different replication methods to scale social network analysis. Hama [5] and Giraph [4] are two open-source projects targeting large graph processing. They adopt Pregel's programming model and their storage is built on top of the Hadoop Distributed File System. While the solutions mentioned above focus on batch processing, there are transactional graph processing databases such as Neo4j and InfiniteGraph. Finally, recently, a number of cloud-based data management systems have been developed for other important workloads such as data warehousing [2,35,77] and on-line transaction processing [22], which are beyond the scope of this chapter.

7.3.2 Comparison of Existing Systems

Table 7.1 provides a brief comparison of a number of representative graph processing systems with respect to their properties of graph storage, support of online processing, main-memory processing and distributed processing. Neo4j and HyperGraphDB

TABLE 7.1
Comparison of Representative Systems (An Extended Version Based on Table 2 in Previous [68])

	Native Graphs	Online Query Processing	Memory-Based Exploration	Distributed Parallel Processing
Neo4j	Yes	Yes	No	No
HyperGraphDB	No	Yes	No	No
InfiniteGraph	Yes	Yes	No	Yes
MapReduce	No	No	No	Yes
PEGASUS	No	No	No	Yes
Surfer	Yes	No	Yes	Yes
Googles Pregel	No	No	No	Yes
Microsofts Trinity	Yes	Yes	Yes	Yes

are two centralized graph processing engines that do not partition data graphs to multiple machines. Both can support online query processing. However, their scalability is limited, because they cannot handle very large graphs efficiently, due to the costly disk accesses. For distributed graph processing systems, many engines are disk-based, mainly for reliability, and scalability. In-memory graph explorations resolve the random I/O bottleneck of Trinity and Surfer. As for graph partitioning, most graph engines (except Surfer) use random hash partitioning by default. Surfer adopts the network performance aware graph partitioning, specifically designed for cloud environments.

To illustrate the differences between these systems with an example, we briefly compare their reported performances of the PageRank computation, which is a typical algorithm for benchmarking graph processing.

Neo4j and HyperGraphDB are centralized graph processing engines, and hence, the graphs that they can process are obviously limited by the centralized server. As graph processing often involves random data accesses, graphs that cannot fit into main memory may incur numerous disk accesses that significantly affects performance.

Pegasus supports iterative matrix–vector multiplications and implemented the matrix approach for computing PageRank and its optimizations. Their experiments confirmed that performance improves as the number of machines increases and performance scales linearly as graph size increases beyond 1 billion nodes. Pegasus finished one PageRank iteration in around 100 seconds on YahooWeb (1.4 billion nodes) under the default setting of 9 supercomputers.

Pregel implemented a vertex-oriented PageRank algorithm under the message passing paradigm. To handle large graph data, its original implementation uses a default random hash function to partition graph data to worker processes. Giraph [4] is a publicly available implementation for Pregel. Its performance of PageRank has been reported in comparison with Trinity [67]. Trinity also implemented a vertex-oriented PageRank algorithm and ran it on eight commodity machines. Trinity keeps graphs in the main memory, which leads to superior efficiency. In particular, Trinity completes one PageRank iteration on a 1 billion node graph in less than 1 minute. In comparison, Giraph is at least two orders of magnitude slower and runs out of memory processing some large graphs.

Surfer tested the Network Rank algorithm* on a social network that consists of more than half a billion nodes and about 30 billion edges. The main focus of Surfer is to show the improvement in performance due to its network performance aware graph partitioning. The speedup of the response time observed from the MapReduce engine built on top of a MapReduce platform ranges from 1.7 to 5.8 times faster than the original response times, under different network topology settings.

7.3.3 Other Graph Processing Platforms/Systems

We have recently seen that cloud computing platforms have been equipped with emerging hardware such as multicore CPUs and GPUs (Graphics Processing Units).

* Network ranking is the generation of a ranking on the vertices in the graph using PageRank or its variants.

Beyond machine-level parallelism, it is desirable to exploit intra-machine parallelism. On multicore CPUs, parallel libraries like MTGL [12] have been developed for parallel graph algorithms. MTGL offers a set of data structures and APIs for building graph algorithms. The MTGL API is modeled after the Boost Graph Library [69] and optimized to leverage shared memory multithreaded machines. The SNAP framework [7] provides a set of algorithms and building blocks for graph analysis, especially for small-world graphs. On the GPU, a general-purpose programming framework called Medusa [80] has been developed. The goal is to hide the details of graph programming and GPU runtime from users. In contrast to Pregel, Medusa adopts very fine-grained processing on vertices/edges/messages to exploit the massive parallelism of the GPU. Additionally, there are specific parallel graph algorithms on the GPU [34,36,48,75].

7.4 UNEVEN BANDWIDTH BETWEEN THE MACHINES OF THE CLOUD

The cloud-based solutions discussed in the previous section provide a user-friendly platform for users to develop their custom logic without worrying how the underlying interconnected machines operates. However, the unique network environment that consists such number of servers does further add fuel to the challenges of large graph processing. In this section, we discuss the factors on the cloud (such as hardware and software) that reveal the major factors of network bandwidth unevenness in the cloud.

7.4.1 Factor 1: Network Environment

Due to the significant scale, the cloud network environment is significantly different from those in previous distributed environment [44,46,52], for example, Cray supercomputers or a small-scale cluster. In a small-scale cluster, the network bandwidth is often roughly the same for every machine pair. However, the network bandwidth of the cloud environment is uneven among different machine pairs.

Current cloud infrastructures often use a switch-based tree structure to interconnect the servers [10,32,41]. Machines are first grouped into *pods*, and then pods are connected to higher-level switches. A natural consequence of such a topology is that the network bandwidth of any machine pair is not uniform that is influenced by the switches that connect the two machines [37]. The intra-pod bandwidth is much higher than the cross-pod bandwidth.

The knowledge of network topology (such as multilevel data reduction along the tree topology [23] and partition-based locality optimizations [64]) and scheduling techniques [38] are crucial for advanced optimization in the cloud. However, it should also be remarked that the topology information in the cloud is usually not available to cloud users due to the virtualization and system management issues.

Finally, a simple reason for network unevenness can be that the commodity computers in the cloud may not have a uniform network configuration (e.g., network adaptors). As the cloud evolves, its computers may become heterogeneous from generations to generations [79]. For example, current mainstream network adaptors provide 1 Gb/sec, and the adaptors with 10 Gb/sec has been gradually employed. These

hardware factors result in the unevenness of the network bandwidth among machines in the cloud.

7.4.1.1 Case Study

As discussed, the network bandwidth among different machine pairs can vary significantly. Such network bandwidth unevenness has been observed by cloud providers [10,41]. He et al. [19] have also observed significant network bandwidth unevenness in Amazon EC2. Figure 7.1 shows the network bandwidth of every machine pair among 64 and 128 small instances (i.e., virtual machine) on Amazon EC2. The network bandwidth varies significantly. The mean (MB/sec) and standard deviation are (112.8, 37.5) and (115.0, 40.2) for 64 and 128 small instances, respectively. It is observed that some pairwise bandwidth are very high (e.g., more than 500 MB/sec). The possible reason is that those small instances can be allocated to the same physical machine.

He et al. [19] also note that the network bandwidth between two instances in the public cloud is temporarily stable, with similar results observed in the another study [76]. This allows to develop network performance aware optimizations based on the network bandwidths measured at a particular recent time point.

7.4.2 Factor 2: Virtualization

In addition to hardware factors, software techniques in the cloud can result in network bandwidth unevenness. In particular, virtualization has been a crucial facility of the cloud. It hides the network topology or the real configurations of the machines underneath a cloud system from users. In fact, in cloud environments, users do not have administrator privileges on the hardware under the virtualization layer. A popular optimization in virtualization is virtual machine consolidation, for better resource utilization of virtualization. However, the consolidation process may induce concurrent tasks to compete for the network bandwidth on the same physical machine. Different degrees of consolidation cause the network bandwidth unevenness among physical machines.

7.5 NETWORK BANDWIDTH AWARE GRAPH PARTITIONING TECHNIQUE FOR THE CLOUD

Due to the massive volume of graph data, even a baseline graph processing engine should store a large graph into partitions, as opposed to a single flat storage. However, graph partitioning itself should be effectively integrated into the large processing in the cloud environment. There are a number of challenging issues in such an integration. First, graph partitioning itself is a very costly task, which in particular generates much network traffic. Second, the network bandwidth unevenness described in Section 1.3 affects the way of graph partitioning and graph partition storage on the machines. Since the number of graph partitions and the number of machines for graph processing can be very large, the possible solution space of storing graph partitions to the machines is huge. Consider P partitions to be stored on P machines. The space includes $P!$ possible solutions. Another problem is how to make both the graph

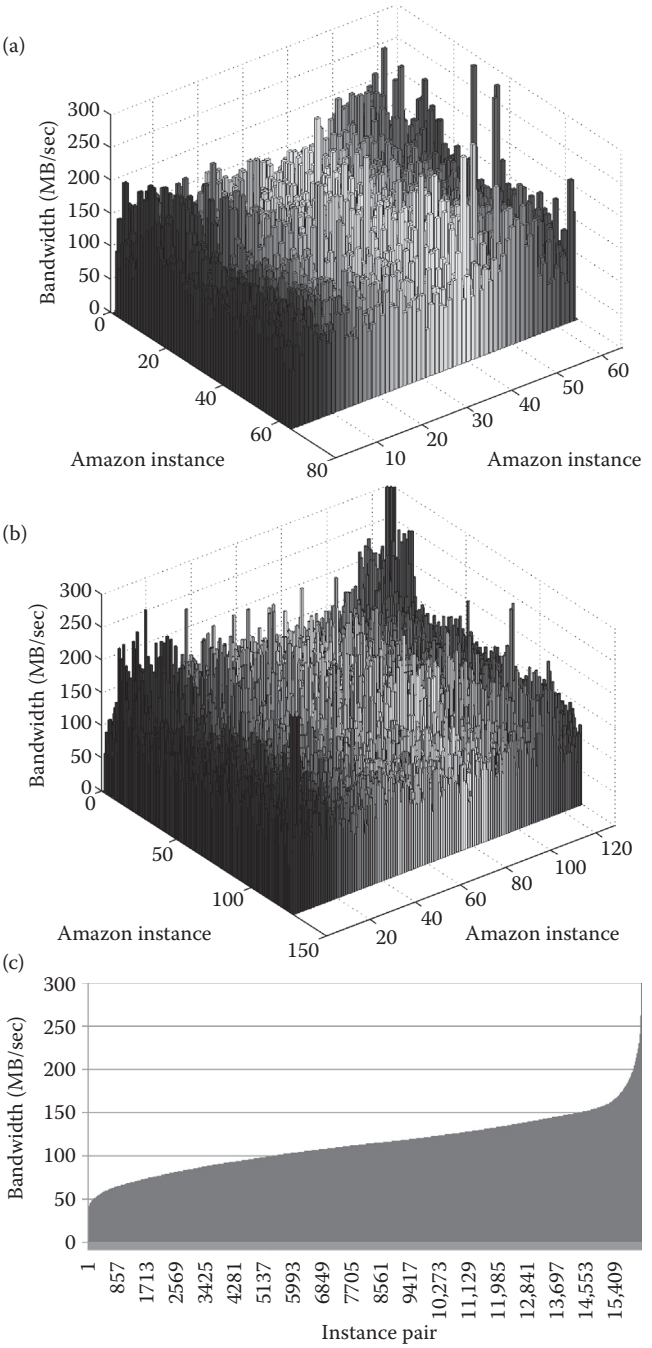

FIGURE 7.1 Network bandwidth unevenness in Amazon EC2: (a, b) pairwise network bandwidth varying the number of small instances, (c) the distribution of pairwise network bandwidth. The y-axis of all figures is capped at 300 for clarity.

partitioning and graph processing algorithm aware of the bandwidth unevenness for networking efficiency.

Foremost, graph partitioning has been a classical combinatorial optimization problem, with an input objective function. The input objective function is to minimize the number of cross-partition edges with the constraint of all partitions with similar number of edges. This is because the total number of cross-partition edges is often a good indicator of the amount of communication between partitions in distributed computation. It is an NP-complete problem [50].

The network performance aware graph partitioning framework discussed in this section improves the network performance of graph partitioning process itself. Moreover, the partitions generated from the framework improve the network performance of graph processing tasks. The basic idea of the framework is to partition, store, and process the graph partitions according to their numbers of cross-partition edges such that the partitions with a large number of cross-partition edges are stored in the machines with high network bandwidth between them, as the network traffic requirement for those graph partitions is high. To achieve this, the framework partitions both the data graph and a "machine graph" (defined next) simultaneously.

7.5.1 Machine Graph

To capture the network bandwidth unevenness, a complete weighted undirected graph (namely *machine graph*) models the machines chosen for graph partitioning. Each machine is modeled as a vertex; an edge represents the connectivity between the two machines, and the bandwidth between any two machines is represented as the weight of an edge. For simplicity, assume that each machine has the same configuration in terms of computation power and main memory. In practice, users usually acquire the virtual machines of the same type for one application, because of convenience and management. In addition, an undirected graph is used in the model, as the bandwidth can often be similar in both directions.

Machine Graph Building. The machine graph can be built without the knowledge or control of the network physical topology, as follows.

Given a set of machines for partitioning, the machine graph can be constructed by calibrating the network bandwidth between any two machines in the set. The network bandwidth can be measured by sending a data chunk of 8 MB and using the average of twenty measurements as the estimated bandwidth. For N virtual machines, only N iterations of calibrations are needed to measure all pairwise performance. In each iteration, $\frac{N}{2}$ machine pairs are calibrated. The maintenance is based on the classic exponential average by getting the bandwidth of data transfer in the graph processing.

Example

The left part of Figure 7.2a illustrates the machine graph for four machines in a cluster with tree topology. The edge thickness represents the weight: a thicker edge means a link with higher bandwidth. The example cluster consists of two

FIGURE 7.2 Mapping on the partition sketches between the machine graph and the data graph. (a) Four machines are chosen. (b) Three machines are chosen.

pods, and each pod consists of two machines. Assuming that the intra-pod network bandwidth is higher than the inter-pod one, and the intra-pod bandwidth is the same across pods, the machine graph consists of four vertices and six edges. The intra-pod connections are represented as thicker edges, indicating that they have a higher interconnected bandwidth.

7.5.2 Partition Sketch

The *process* of a multilevel graph partitioning algorithm is modeled as a tree structure (namely *partition sketch*). Each node in the partition sketch represents the graph acting as the input for the partition operation at a level of the entire graph partitioning process: the root node representing the input graph; nonleaf nodes at level $(i + 1)$ representing the partitions of the ith iteration; the leaf nodes representing the graph partitions generated by the multilevel graph partitioning algorithm. The partition sketch is a k-ary tree for k-section-based graph partitioning algorithm. In practice, graph partitioning is often done using bisections iteratively, and hence, the partition sketch is represented as a binary tree. If the number of graph partitions is P, the number of levels of the partition sketch is $(\lceil \log_2 P \rceil + 1)$.

Example

Figure 7.3 illustrates the correspondence between partition sketch and the bisections in the entire graph partitioning process. In the figure, the graph is divided into four partitions, and the partition sketch has three levels.

7.5.2.1 Design Principles of Ideal Partition Sketch

Among various partition sketches, an *ideal partition sketch* describes the partitioning process that strikes a balance between partitioning time and partition quality.

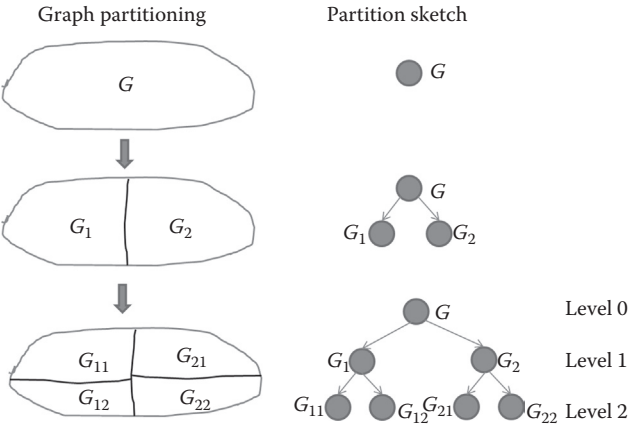

FIGURE 7.3 Correspondence between bisections and the partition sketch for the process of partitioning the graph into four partitions.

The ideal partition sketch represents the iterative partition process with the optimal bisection on each partition. In each bisection, the optimal bisection minimizes the number of cross-partition edges between the two generated partitions. This is the best case that existing bisection-based algorithms [44,46,47] can achieve. Partitioning with optimal bisections does not necessarily result in partitions with the globally minimum number of cross-partition edges. However, existing studies [44,46] have demonstrated that they can achieve relatively good partitioning quality, approaching the global optimum. Furthermore, the ideal partition sketch exhibits a few interesting properties:

7.5.2.1.1 Local Optimality

Denote $C(n_1, n_2)$ as the number of cross-partition edges between two nodes n_1 and n_2 in the partition sketch. Given any two nodes n_1 and n_2 with a common parent node p in the ideal partition sketch, we have $C(n_1, n_2)$ is the minimum among all the possible bisections on p.

By definition of the ideal partition sketch, the local optimality is achieved on each bisection.

7.5.2.1.2 Monotonicity

Suppose the total number of cross-partition edges among any partitions at the same level l in the partition sketch to be T_l. The monotonicity of the ideal partition sketch is that $T_i \le T_j$, if $i \le j$.

Proof (sketch). According to multilevel graph partitioning, a cross-partition edge in level i is still a cross-partition edge in level $i + 1$. Additionally, more cross-partition edges are created during the bi-section at level i. Thus, the set of cross-partition edge in level i is a subset of that in level $i + 1$. Thus, $T_i \le T_{i+1}$. Therefore, $T_i \le T_j$, if $i \le j$. □

The monotonicity reflects the increase in the number of cross-partition edges in the recursive partitioning process.

7.5.2.1.3 Proximity

Given any two nodes n_1 and n_2 with a common parent node p, any other two nodes n_3 and n_4 with a common parent node p', and p and p' are with the same parent, we have $C(n_1, n_2) + C(n_3, n_4) \ge C(n_{\pi(1)}, n_{\pi(2)}) + C(n_{\pi(3)}, n_{\pi(4)})$ where π is any permutation on (1, 2, 3, 4).

Proof (sketch). According to local optimality, we know that $C(p, p') = C(n_1, n_3) + C(n_1, n_4) + C(n_2, n_3) + C(n_2, n_4)$ is the minimum. Thus, we have

$$C(n_1, n_2) + C(n_1, n_4) + C(n_3, n_2) + C(n_3, n_4) \ge C(p, p') \tag{7.1}$$

$$C(n_1, n_2) + C(n_1, n_3) + C(n_4, n_2) + C(n_4, n_3) \ge C(p, p') \tag{7.2}$$

Substituting $C(p, p')$, we have

$$C(n_1, n_3) + C(n_2, n_4) \le C(n_1, n_2) + C(n_3, n_4) \tag{7.3}$$

$$C(n_2, n_3) + C(n_1, n_4) \le C(n_1, n_2) + C(n_3, n_4) \tag{7.4}$$

That means, we have $C(n_1, n_2) + C(n_3, n_4) \geq C(n_{\pi(1)}, n_{\pi(2)}) + C(n_{\pi(3)}, n_{\pi(4)})$ where π is any permutation on (1, 2, 3, 4). □

The intuition of the proximity is, at a certain level of the ideal partition sketch, the partitions with a low common ancestor have a larger number of cross-partition edges than those with a high common ancestor.

These properties of the partitioning sketch indicate the following design principles for graph partitioning and processing, to match the network bandwidth with the number of cross-partition edges.

P_1. Graph partitioning and processing should gracefully adapt to the bandwidth unevenness in the cloud network. The number of cross-partition edges is a good indicator on bandwidth requirements. According to the local optimality, the two partitions generated in a bisection on a graph should be stored on two machine sets such that the total bandwidth between the two machine sets is the lowest.

P_2. The partition size should be carefully chosen for the efficiency of processing. The number of partitions should be no smaller than the number of machines available for parallelism. According to the monotonicity, a small partition size increases the number of levels of the partition sketch, resulting in a large number of cross-partition edges. On the other hand, a large partition may not fit into main memory of a machine, which results in random disk I/O in accessing the graph data.

P_3. According to proximity, the nodes with a low common ancestor should be stored together in the machine sets with high interconnected bandwidth to reduce the performance impact of the large number of cross-partition edges.

7.5.3 Bandwidth Aware Graph Partitioning

The network bandwidth aware framework for graph partitioning and processing in the cloud exploits the ideal partition sketch and the machine graph discussed in Sections 7.5.1 and 7.5.2, which enhances a popular *multilevel graph partitioning algorithm* with the *network performance awareness*. This subsection presents some design issues and an overview of such a framework.

7.5.3.1 Background on the Bisection in the Multilevel Graph Partitioning

Since graph bisection has been a key operation in multilevel graph partitioning [44,46], we briefly introduce the process of bisection. There are three phases in a graph bisection, namely *coarsening*, *partitioning*, and *uncoarsening*, as illustrated in Figure 7.4. The coarsening phase consists of multiple iterations. In each iteration, multiple adjacent vertices in the graph are coarsened into one according to some heuristics, and the graph is condensed into a smaller graph. The coarsening phase ends when the graph is small enough, in the scale of thousands of vertices. The partitioning phase divides the coarsened graph into two partitions

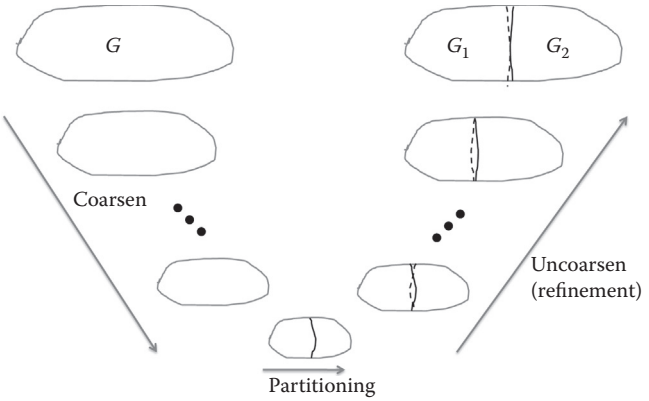

FIGURE 7.4 The three phases in graph bisection: *coarsening*, *partitioning*, and *uncoarsening.*

using a sequential and high-quality partitioning algorithm such as GGGP (Greedy Graph Growing Partitioning) [45]. In the uncoarsening phase, the partitions are then iteratively projected back toward the original graph, with a local refinement on each iteration.

The iterations are highly parallelizable, and their efficiency and scalability has been evaluated on shared-memory architectures (such as Cray supercomputers) [44,46]. However, in the coarsening and uncoarsening phases, all the edges may be accessed, generating a lot of network traffic if the input graph is stored in distributed machines.

7.5.3.2 Network Transfer due to Cross Edges

Given a set of machines to partition the graph, the graph is initially stored in those machines (usually according to the simple hash function). At each bisection, all edges and vertices are accessed multiple times for coarsening and uncoarsening. It generates a lot of network traffic. Thus, bisection should be designed to be aware of the network bandwidth unevenness.

Assume that the amount of network traffic sent along each cross-partition edge is the same (denoted as b). Denote the number of cross-partition edges from partition G_i to G_j to be $C(G_i, G_j)$, and the network bandwidth between the machines stored G_i and G_j to be $B_{i,j}$. Since network bandwidth is a scarce resource in the cloud environment [23,37], the bandwidth can be considered as the main indicator for network performance, and it approximates the network data transfer time from G_i to G_j to be $\frac{c(G_i, G_j) \times b}{B_{i,j}}$. This approximation is sufficient for large graph processing in both private and public cloud environments. Assuming P graph partitions are stored on P different machines, the total network data transfer time incurred in all partition pairs is $\sum_{i=0}^{P-1} \sum_{j=0}^{P-1} \frac{C(G_i, G_j) \times b}{B_{i,j}}$.

Clearly, if the network bandwidth among different machine pairs ($B_{i,j}$, $\forall i, j < P$) is constant, minimizing the total number of cross-partition edges also minimizes the total network data transfer time.

7.5.3.3 Partitioning the Machine Graph

An observation on multilevel graph partitioning algorithms is that *due to the divide-and-conquer nature, there is no data exchange between the two bisection subpartitions generated from the same bisection*. Suppose a distinct subset of machines is responsible for each of the two subpartitions. The network connections between the two subsets of machines are no longer involved in the deeper levels of the bisection. That means, the partitioning algorithm should pick the high bandwidth connections remaining in the subset of machines, and leave the low bandwidth connections as those between the two subsets of machines. This is analogous to performing graph partitioning on the machine graph with respect to minimizing the total bandwidth between two subsets of machines. That results in the correspondence between partitioning the data graph and partitioning the machine graph, and the algorithm gradually assigns the subset of machines that are suitable to handle graph partitioning at a certain level.

7.5.3.4 Network Performance Aware Partitioning

Putting these together, the algorithm traverses the partition sketches of the machine graph and the data graph and builds a mapping between the machines and the partitions. At each level of graph partitioning, the framework partitions the data graph and machine graph *simultaneously* and matches the network bandwidth in the cloud to the number of cross-partition edges according to the partition sketch and the machine graph. The mapping guides the machines where the graph partition is further partitioned, and where the graph partition is stored. At the leaf level, graph partitions are stored in the machine in the corresponding node in the machine graph. Finally, the partition sketches for both machine graph and data graph are generated.

Example

Figure 7.2 illustrates the mapping between two machine graphs and a data graph for the partitioning framework. Take case (a) where four machines are selected as an example. The bisection on the entire graph G is done on all the four machines. At the next level, the bisections on G_1 and G_2 are performed on pods M_1 and M_2, respectively. Finally, the partitions are stored in the machines according to the mapping.

Regarding the local graph partitioning algorithm, any classical graph partitioning algorithms such as Metis [47] can be used. For example, Metis can be used to partition the machine graph, since the machine graph can often fit into the main memory of a single machine. On the bisection of the machine graph, the objective function is to minimize the weight of the cross-partition edges with the constraint of two partitions having around the same number of machines. This objective function matches the bandwidth unevenness of the selected machines. The goal

of minimizing the weight of cross-partition edges in the machine graph corresponds to minimizing the number of cross-partition edges in the data graph. This is a graceful adaptation on assigning the network bandwidth to partitions with different number of cross-partition edges.

7.5.3.5 Partition Numbers

A minor detail is that it is preferable to make partitions with the roughly same number of machines is for load-balancing purpose, since partitions in the data graph also have similar sizes. On the other hand, the number of partitions returned by the algorithm may also be specified by the user. A reasonable choice is to determine P so that each graph partition can fit into the main memory of a machine. This is to avoid the significant performance degradation due to the random disk I/O in graph processing.

Finally, the partitioning algorithm discussed in this section satisfies the three design principles: (1) the number of cross-partition edges is gradually adapted to the network bandwidth. In each bisection of the recursion, the cut with the minimum number of cross-partition edges in the data graph coincides that with minimum aggregated bandwidth in the machine graph. (2) The partition size is tuned according to the amount of main memory available to reduce the random disk accesses. (3) In the iteration, the proximity among partitions in the machine graph matches that in the data graph.

7.6 HIERARCHICAL COMBINATION OF EXECUTION

With partitioned graph, the graph execution model may exploit data locality to reduce network traffic in data-intensive computing systems [23,37]. The basic idea is to apply a *Combine*() function (i.e., *Combiner* in Pregel), and perform partial merging of the intermediate data before they are sent over the network. Combination is applicable when the combination function is annotated as an associative and commutative function.

A basic approach is *local combination*. Current cloud-based graph engines like Pregel and Trinity support this basic approach. For all the graph partitions on a machine, one may apply the local combination on the boundary vertices belonging to the same remote partition and send the combined intermediate results back to the local partition for further processing.

Local combination is not aware of the network bandwidth unevenness in the cloud network environment. This motivates the approach of *hierarchical combination*. In local combination, it requires network data transfer for the boundary vertices of the graph partition. Due to the irregular graph structures, the source vertices are likely to be scattered on many different machines. Thus, many data transfers are performed on the relatively low-bandwidth machine pairs caused by the network bandwidth unevenness. Therefore, instead of direct data transfers after local combination, one may exploit the machine graphs for local combination as follows.

The data of the source vertices can be combined among the machines with high bandwidth before sending them to the target machine via the connections with low bandwidth. Hierarchical combination applies this idea in multiple levels according to

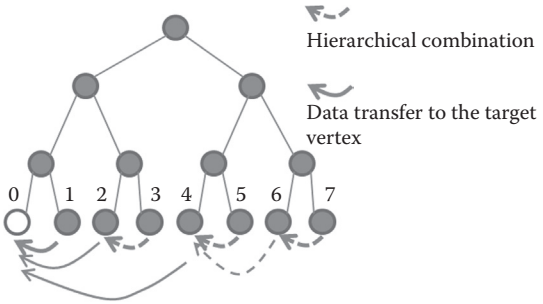

FIGURE 7.5 Hierarchical combination according to the partition sketch of the machine graph of eight machines.

the partition sketch of the machine graph. With the hierarchical combination optimization, the data transfer on the low-bandwidth connection is reduced.

Example

Figure 7.5 illustrates one example of performing hierarchical combination on eight machines. Suppose each machine holds one graph partition and machine 0 needs to read data from other machines. Note that the partition sketch of the machine graph has captured the network bandwidth unevenness. After local combination on each machine, the first-level combination between two machines (for example, between machines 6 and 7) are performed, and the result is stored on a *representative* machine. Let us assume machines 2, 4, and 6 are the representative machines at the first-level combination. Further combination is performed on the representative machines. Finally, all the partial results are sent to machine 0. On the low-bandwidth connections between machine 0 and machine i ($4 \leq i \leq 7$), hierarchical combination has only one data transfer for the partial results, compared with four in the baseline implementation with local combination.

7.7 RELATED WORK ON GRAPH PARTITIONING

In addition to the recent work on network performance aware partitioning, a large number of graph partitioning techniques have been proposed. In this section, we provide a review of general graph partitioning algorithms and then highlight some existing distributed graph partitioning algorithms.

Graph partitioning is important not only in emerging applications like Web and social graphs, as discussed in previous sections, but also traditional applications such as circuit placement and matrix factorization. The problem is NP-hard for general graphs [15,29]. There have been many studies on graph partitioning problems, which we can divide into five major categories: geometric methods [11,26,60], spectral methods [25,71], multilevel methods [44,46], metaheuristic-based approaches [51], and streaming graph partitioning [3,72,74]. We also refer the readers to two comprehensive surveys [28,51] for more related work on graph partitioning.

7.7.1 Geometric Methods

In geometric methods, each node of a graph is associated with a geometric location (or coordinate). A classic example of a geometric algorithm is recursive coordinate bisection [11]. This can be efficiently implemented with the multidimensional binary tree or kD tree data structure. Follow-up research studies (e.g., [26,60]) have improved the classic methods with more advanced heuristics or bisection methods.

7.7.2 Spectral Methods

Spectral graph theory studies the relationships of fundamental properties of graphs (e.g., algebraic connectivity) and the eigenvectors and eigenvalues of the Laplacian matrix associated with the graphs [14,20]. In particular, the eigenvector associated with algebraic connectivity (also known as the Fiedler vector) can be used to partition graphs. There have many proposals on spectral partitioning and spectral bisection. With spectral bisections, one can incorporate them into multilevel graph partitioning. It should be remarked that most of the results from spectral graph theory are specific to undirected graphs.

An advantage of the spectral techniques is that they are supported by industrial-strength softwares, not to mention the availability of advanced optimizations.

7.7.3 Metaheuristic-Based Approaches

In general, it is difficult to produce high-quality solutions with approximation algorithms of theoretical bounds. The metaheuristic approaches have mostly concentrated on finding high-quality solutions without performance guarantee in a reasonable amount of time. Representatives of metaheuristic approaches include simulated annealing [39], tabu search [66], ant colony optimization [16], and genetic algorithms [57]. More details of these algorithms can be found in the survey [51].

7.7.4 Streaming Graph Partitioning

Instead of optimizing the graph partitioning quality, streaming graph partitioning emphasizes the graph partitioning performance while achieving a much better graph partitioning quality than random hashing. It usually requires a simple pass (or scan) of the graph, and generates the graph partitioning during the scan. Due to the streaming nature, this category of graph partitioning algorithm can also be applicable to streaming graphs. For online streaming graphs, Aggarwal et al. [3] proposed an algorithm for clustering graph streams. They used a hash-based compression of the edges to create microclusters onto a smaller domain space. They showed that their method provides bounded accuracy in terms of distance computations. Stanton et al. [72] developed simple heuristics for streaming graph and demonstrated their effectiveness against simple hashing and Metis schemes. Fennel [74] is a general framework for streaming graph partitioning. All these studies have demonstrated significant gains in terms of the communication cost and runtime.

7.7.5 Distributed Graph Partitioning Algorithms

Prior to the network bandwidth aware framework described in the previous sections, distributed graph partitioning [24,47,52] was the traditional way of reducing data shuffling in distributed graph processing. The commonly used distributed graph processing algorithms are multilevel algorithms [44,46,73], which are also used in the partitioning algorithm described in this chapter. They have been proved efficient in many applications.

7.7.6 The Metis Framework

A highly popular tool Metis [47] is a multilevel graph partitioning framework whose implementation is fast, robust, and easy to use. The multilevel graph partitioning framework contains three phases: (1) "coarsening by maximal match until the graph is small enough"; (2) partitioning the coarsest graph by any reasonable partition algorithm; and (3) refining the partitions by vertex swapping algorithm. ParMetis [44,46] is a parallel multilevel graph partitioning algorithm, with a minimum bisection on each level. It has been demonstrated to perform very well on shared-memory architectures [46]. Additionally, various different heuristics have been proposed for the quality of coarsening and refinement (e.g., [9,59]).

7.8 OPEN PROBLEMS

Despite recent efforts in large-graph processing in the cloud, many open problems remain to be explored in future [68]. As suggested by Shao et al., these open problems include architectural design, application needs, computation model, and ownership. We briefly elaborate on the problems as follows.

7.8.1 Architectural Design

Currently, in-memory processing is the key technique for resolving the random I/O in graph processing (besides algorithmic design). However, as the increasing popularity of graph-centric applications (such as the fast growing social graphs and web graph), it is yet to confirm whether the in-memory solution is the most favorable system design in terms of performance, energy consumption, and total ownership cost, among other things. Thus, in addition to main-memory based solutions, one may investigate other emerging storages such as solid state drives (SSD), which also exhibits much faster random I/O speed than hard disks. A hybrid storage system of SSD and main memory may also be possible for increasingly large graphs. More research work is required on efficient data structures and algorithms for graphs on such a hybrid system.

7.8.2 Application Needs

Web and social networks have been the two main driving applications for graph processing. Their application needs evolve from offline to online processing. Many

application needs, such as data consistency and transaction management, have received relatively little research attention. These needs are already difficult problems in the context of flat data-like distributed relational databases [8,21], and they will be more challenging for graph systems.

7.8.3 Computation Model

"One size does not fit all." Application needs drive the computation model. Current systems are mainly based on MapReduce and the vertex oriented execution model. However, it is an open problem to extend these models with indexes and different application needs such as consistency and transaction management.

7.8.4 Cost of Ownership

Ideally, users want to minimize the cost of ownership while satisfying the performance requirement and other quality of service attributes. However, the design space is huge for various different hardware and software components. As specific to the cloud, different cloud providers offer very different price structures. Even for the same cloud provider, the capabilities of virtual machines can be quite different [27]. More research has to be conducted on automatic and customizable design for the cost of ownership.

7.9 SUMMARY

In this chapter, we have surveyed a number of applications of large graphs and existing representative cloud-based large graph processing systems. One of the classic techniques for handling large graphs is graph partitioning. The chapter reviewed the network unevenness of the cloud, which poses new challenges to graph partitioning techniques. In particular, networks with high bandwidth between machines can process more tasks on cross-partition edges. This chapter then focused on network performance aware graph partitioning. The techniques include modeling machines and the network bandwidth between them as a machine graph, and partitioning the graph corresponding to the machine graph. These techniques minimize network traffic in both partitioning and processing. The processing on partition graphs may further exploit the locality of the partitions to reduce communications. There are many open problems that require more research efforts in this field.

REFERENCES

1. A new application award: Semantic web challenge. http://challenge.semanticweb.org/, 2013.
2. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endow.*, 2009.
3. C. C. Aggarwal, Y. Zhao, and P. S. Yu. A framework for clustering massive graph streams: Submission to best of SDM 2010 issue. *Stat. Anal. Data Min.*, 3(6):399–416, December 2010.
4. Apache Giraph. http://giraph.apache.org/.

5. Apache Hama. http://hama.apache.org/.
6. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *6th International Semantic Web Conference*, pages 11–15, 2007.
7. D. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS*, pages 1–12, 2008.
8. P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Hat, not cap: Highly available transactions. *CoRR*, abs/1302.0309, 2013.
9. U. Benlic and J.-K. Hao. An effective multilevel memetic algorithm for balanced graph partitioning. In *ICTAI* (1), pages 121–128, 2010.
10. T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.
11. M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multi-processors. *IEEE Trans. Comput.*, 36(5):570–580, May 1987.
12. J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS*, pages 1–14, March 2007. IEEE.
13. Bing Index Team. Understand your world with Bing. http://hk.bing.com/blogs/site blogs/b/search/archive/2013/03/21/satorii.aspx, 2013.
14. A. E. Brouwer and W. H. Haemers. *Spectra of Graphs*. Springer, 2012.
15. T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3):153–159, May 1992.
16. T. N. Bui and L. C. Strite. An ant system algorithm for graph bisection. In *GECCO*, pages 43–51, 2002.
17. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.
18. R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, SIGMOD'10*, pages 1123–1126, New York, 2010. ACM.
19. R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, pages 3:1–3:13, 2012.
20. F. Chung. Spectral Graph Theory. Conference Board of the Mathematical Sciences, 1997.
21. J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.
22. S. Das, D. Agrawal, and A. El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *SoCC: ACM Symposium on Cloud Computing*, 2010.
23. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
24. B. Derbel, M. Mosbah, and A. Zemmari. Fast distributed graph partition and application. In *IPDPS*, 2006.
25. W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM J. Res. Dev.*, 17(5):420–425, September 1973.
26. C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int. J. Numer. Methods Eng.*, 36(5):745–764, 1993.
27. B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing, SoCC'12*, pages 20:1–20:14, New York, 2012. ACM.

28. P.-O. Fjallstrom. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 1998.
29. M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, STOC'74*, pages 47–63, New York, 1974. ACM.
30. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
31. D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.
32. C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A scalable and fault-tolerant network structure for data centers. *SIGCOMM*, 38(4):75–86, 2008.
33. Hadoop. http://hadoop.apache.org/.
34. P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing, HiPC'07*, pages 197–208, 2007. Springer-Verlag.
35. B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10*, pages 63–74, New York, 2010. ACM.
36. G. He, H. Feng, C. Li, and H. Chen. Parallel SimRank computation on large graphs with iterative aggregation. In *SIGKDD*, 2010.
37. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.
38. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.
39. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation. Part I, graph partitioning. *Oper. Res.*, 37(6):865–892, October 1989.
40. N. Kallen, R. Pointer, J. Kalucki, and E. Ceaser. Github. https://github.com/twitter/flockdb, 2013.
41. S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements and analysis. In *IMC*, 2009.
42. U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop. Technical Report CMU-ML-08-117, CMU, 2008.
43. U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system–implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM'09*, pages 229–238, 2009.
44. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, Supercomputing'96*, 1996. IEEE Computer Society Press.
45. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
46. G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.
47. Karypis Lab. *Family of Graph and Hypergraph Partitioning Software*, 2013.
48. G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Graphics Hardware*, pages 47–55, 2008.
49. KEGG Laboratories. KEGG: Kyoto encyclopedia of genes and genomes. http://www.genome.jp/kegg/kegg1.html, 2013.
50. B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(1):291–307, 1970.

51. J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon. Genetic approaches for graph partitioning: A survey. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO'11*, pages 473–480, New York, 2011. ACM.
52. S. Koranne. A distributed algorithm for k-way graph partitioning. In *EUROMICRO*, 1999.
53. J. Leskovec. Stanford network analysis project. http://snap.stanford.edu/, 2013.
54. LinkedIn. Numbers of LinkedIn members as of 1st quarter 2013. http://www.statista.com/statistics/198224/quarterly-member-numbers-of-linkedin/, 2013.
55. Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.
56. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.
57. H. Maini, K. Mehrotra, C. Mohan, and S. Ranka. Genetic algorithms for graph partitioning and incremental graph partitioning. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, Supercomputing'94*, pages 449–457, Los Alamitos, CA, 1994. IEEE Computer Society Press.
58. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD'10*, pages 135–146, 2010.
59. J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In *Proceedings of the 6th International Conference on Experimental Algorithms*, *WEA'07*, pages 242–255, Berlin, Heidelberg, 2007. Springer-Verlag.
60. G. L. Miller, S. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. Technical Report, Ithaca, NY, 1992.
61. NCBI. PubChem. http://pubchem.ncbi.nlm.nih.gov, 2013.
62. NCBI. Submit data to NCBI. http://pubchem.ncbi.nlm.nih.gov/search/#, 2013.
63. L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, 1999.
64. R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*, pages 1–14, 2010.
65. J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM'10*, pages 375–386, 2010.
66. E. Rolland, H. Pirkul, and F. Glover. Tabu search for graph partitioning. Annals of Operations Research, 1996.
67. B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical report, Microsoft Research, 2012.
68. B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: Systems and implementations. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD'12*, pages 589–592, New York, 2012. ACM.
69. J. Siek, L.-Q. Lee, and A. Lumsdaine. The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, 2002.
70. Socialbakers. Facebook pages statistics. http://www.socialbakers.com/facebook-pages/, 2011.
71. D. Spielman. Spectral graph theory and its applications. In *48th Annual IEEE Symposium on Foundations of Computer Science, 2007. FOCS'07*, pages 29–38, 2007.
72. I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'12*, pages 1222–1230, New York, 2012. ACM.

73. A. Trifunović and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68, May 2008.
74. C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. Technical Report MSR-TR-2012-113, Microsoft Research, 2012.
75. V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, 2008.
76. G. Wang and T. S. E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *Proceedings of the 29th Conference on Information Communications, INFOCOM'10*, pages 1163–1171, 2010.
77. J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD'10*, pages 591–602, 2010.
78. W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*, pages 481–492, 2012.
79. M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 321–330, 2008.
80. J. Zhong and B. He. Medusa: Simplified graph processing on GPUS. *IEEE TPDS*, 2013.
81. A. Zhou, W. Qian, D. Tao, and Q. Ma. DisG: A distributed graph repository for web infrastructure (invited paper). In *International Symposium on Universal Communication*, 0:141–145, 2008.