IJICTR International Journal of Information & Communication Technology Research

Volume 6- Number 3-Summer 2014 (53-61)

A Review of the Distributed Methods for Large-Scale Social Network Analysis

Mohsen Kahani
Dept. of Computer Engineering
Ferdowsi University of Mashhad
Mashhad, Iran
kahani@um.ac.ir

Saeid Abrishami
Dept. of Computer Engineering
Ferdowsi University of Mashhad
Mashhad, Iran
s-abrishami@um.ac.ir

Fattane Zarrinkalam
Dept. of Computer Engineering
Ferdowsi University of Mashhad
Mashhad, Iran
fattane.zarrinkalam@stu-mail.um.ac.ir

Received: February 22, 2014 - Accepted: November 24, 2014

Abstract— **Social Network Analysis (SNA) is aimed at studying the structure of a social network, usually represented as a graph, in order to extract the hidden knowledge about the activities and relationships of the users. With exponential increase in the volume and velocity of the data created in today's social networks like Facebook and Twitter, a main requirement for social network analysis is employing computationally efficient algorithms and methods. Since sequential and centralized approaches are far from the desired scalability, a natural solution is to distribute graph of the network on a number of processing machines and perform the execution in parallel. In this paper, existing works on distributed large-scale graph processing are reviewed in four categories regarding their computational model. It is concluded that none of the existing categories outperforms other ones significantly, and therefore no single category addresses the requirements of all different graph algorithms. This highlights the need to research on identifying the types of algorithms for which each category of the computational models is more suitable, and also on how to customize the model for the corresponding type.**

Keywords- Network Analysis; Distributed and parallel processing; Large-Scale Graph processing

I. INTRODUCTION

Social networks are one of the major success stories that have pushed the user-centric nature of Web 2.0 to its limits by enabling users with wide range of social, cultural and educational differences to get involved in content generation and interaction on the web. The large volume of data created through users' activities has made social networks a valuable source of knowledge. Consequently, in the last decade, much research is focused on social network analysis to extract knowledge from the structure of the social networks [1][2].

Generally, two types of methods can be observed in the social network analysis domain [2][3]: 1) egocentric analysis which focuses on how individual entities affect, through their relationships, the larger groups of entities or fragments of the network, and 2) socio-

centric analysis which has a macroscopic point of view and concentrates on studying groups of entities and their relationships, how they evolve, and how they affect each other.

Despite its interesting advantages, success of social network analysis is also hindered by different challenges, a major one of which is the huge amount of data generated with a fast pace on the social networks [1][5]. For instance, in the case of Facebook, the number of users has exceeded 1.2 billion users in 2014, the number of friend connections is more than 150 billion, and the size of Facebook's Graph Search database is greater than 700 terabytes[1].

A social network is usually modeled as a very large graph with vertices and edges representing the network entities and their relationships, correspondingly [2]. As a result, a social network analysis task is usually transformed to a kind of graph processing problem, which requires scalable algorithms and methods.

A possible approach to address this issue is using graph databases which are inherently designed to store and analyze graph data [4]. However, they are mostly based on sequential and centralized algorithms that do not provide the required efficiency when processing very large graphs. For instance, Neo4j[2] is a popular and high-performance graph database. If the graph is not very large, Neo4j stores a large part of data in the main memory, leading to a low response time for query execution on the graph. However, for very large graphs not fitting in the main memory, portions of the graph are stored on disk, and therefore, performance degrades due to the frequent disk access [5].

A more promising approach is to distribute the graph on a number of machines and execute the processing using distributed and parallel algorithms [5][6]. The goal of this paper is to review existing works focusing on distributed processing of large-scale graphs. Two major issues generally addressed by these works are 1) how to efficiently distribute graph data onto processing machines in order to reduce data communication across the underlying network, and 2) how to design concurrent and efficient algorithms for processing the distributed graph data.

This paper is organized as follows. In Section II some required background is described. Section III reviews existing works and discusses their merits and pitfalls. Finally, Section IV concludes the paper by summarizing the review results.

II. Background

A major step in most of the works reviewed in this paper is distributing graph data based on some distribution model. A second step is to execute distributed and parallel algorithms on this distributed data layer.

In the following, first, the issue of data distribution, and then, challenges of the second step are described in more details.

A. Graph Data Distribution

An approach generally used for distributing graph data is based on a network of multiple commodity machines to provide the required processing power. Each machine has access to its own data and if some data is needed that is stored on a remote machine, it is provided via communicating the remote machine across the network.

An advantage of this architecture is that due to the use of commodity hardware, it is cost-effective and therefore horizontally scalable. However, this is obtained at the cost of more complex programming model [1][7].

The main issue in this approach is how to distribute the data over machines in order to reduce the network communication overhead. Regarding this issue, three main models are employed in the existing works:

- A copy of the entire graph is stored on each machine. This model is effective, especially for algorithms that can be divided into independent tasks, since the machines do not need to communicate with each other in order to traverse the graph and collect the required fragments [11]. In order for this model to be effective, it is required to store the whole graph in the main memory of the machine, to eliminate the disk access latency issue. Therefore, the main disadvantage of this model is its misuse of memory, because each machine should have enough memory to store the entire graph. This is a major practical limitation when processing large graphs.
- The input graph is partitioned over different machines, so that each machine stores some portions of the graph. The benefit of this approach is that it enables the processing very large graphs by increasing the number of the processing commodity machines. On the other hand, this approach has the disadvantage that it might need considerable network communication for a single machine to traverse the entire graph or a large subgraph of it. This communication overhead is in contrast with scalability. Therefore, the main issue in this model is how to effectively partition the graph in order to reduce or minimize network communications [16][18][25][26].
- A hybrid model is employed in which the input graph is first partitioned over machines, and a caching mechanism is used to reduce data communications with remote machines [9][14][32]. This leads to an

[1] http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats

[2] www.neo4j.org

improved performance, until the overhead of cache consistency-preserving is kept small.

B. Challenges

Although employing distributed and parallel algorithms improves scalability and efficiency of graph processing tasks, it also introduces some challenges related to the common the characteristics of most graph algorithms. These characteristics include [7]:

- Data-Driven computations: In many graph algorithms, the computational steps are driven by the structure of the underlying graph instance. In other words, it cannot be determined a priori how the computations are distributed across the graph vertices and edges, and it is different for each graph instance. This is in contrast with many matrix computation algorithms, e.g. matrix multiplication, where the execution pattern of the computational steps of the algorithm can be clearly determined regardless of the values of the matrix entries. As a result, it is challenging to find out how the underlying computational steps of the graph algorithm should be partitioned and parallelized.
- Unstructured Data: From a practical point of view, many real world graphs are much irregular and unstructured. As a result, simple schemas for graph partitioning might lead to unbalanced workloads on different processing machines, and therefore poor resource utilization. On the other hand, finding an efficient schema for partitioning graph data across processing machines is much challenging, since it is dependent on the structure of the graph instance, which is not known before the execution time.
- Poor Locality: Due to the unstructured and irregular nature of a graph, and the difficulty of effective graph partitioning on processing machines, it might happen that many graph fragments required by a processing machine are not already stored locally and they must be obtained from a remote machine through costly network communication. This poor locality which is observed in many graph algorithms reduces performance.
- Communication to computation ratio: In many graph algorithms, a processing machine may need to browse through the graph, and hence, the ratio of communications (e.g. for accessing remotely stored graph data) to actual computations is considerably high. This leads to low performance and poor utilization of computational power.

The characteristics and challenges described have been the subject of much research. In the next section, some of the related works are reviewed.

III. Distributed Methods for Larg-scale Gragh Processing

There are many works which propose distributed methods for processing large graphs. These works can be categorized into four groups based on the underlying computational model. Next, these groups are described and compared to each other.

A. MPI-based

MPI (Message Passing Interface) defines a standard interface for a wide range of applications which require a message passing schema [8]. There are some works which propose MPI based methods for processing large-scale graphs. While they generally provide high performance, they also have some problems. For instance, they do not address fault tolerance and it is left to the programmer to implement the required mechanisms. The programmer is also responsible for much of the work regarding scheduling, low-level synchronization and communication [6].

For instance, PBGL [9] is a general MPI based library which supports definition of various distributed graph data structure with different characteristics. It also provides generic algorithms that can be effectively run on different types of graphs. The goal of this library is to improve flexibility and performance, simultaneously. For instance, in order to address the poor-locality challenge mentioned in the previous section, PBGL uses a hybrid model for distributing graph data. This model introduces the concept of *ghost cells* to cache graph data obtained from remote machines. While this usually improves performance, but it also might lead to scalability issues when the number of remote references exceeds a specific threshold. For instance, experiments in [7] have shown this scalability problem for very large-scale graphs.

As another example, Combinatorial BLAS [13] is a distributed library based on MPI for analyzing and mining graph data. It uses a distributed sparse adjacency matrix for representing graph data, and provides a set of primitives based on linear algebra for implementing distributed graph algorithms. It provides flexibility in the sense that it supports modification and customization of the underlying matrix representation format with low cost and complexity.

Compared to PBGL, Combinatorial BLAS provides better scalability by employing more sophisticated graph distribution schema [13]. Actually, it uses two-dimensional distribution by dividing the underlying matrix to equal blocks, i.e. sub-matrices, while PBGL uses one-dimensional distribution by distributing graph only based on its vertices.

B. MapReduce-based

MapReduce is a distributed programming model for processing large volumes of data over a cluster of machines. The input data is initially divided into smaller chunks and distributed over machines. Then, in the map stage, a map function is used to produce intermediate key-value pairs to be fed to the reduce stage. In the reduce stage, a specific computation is executed on the input key-pair values to generate the results, which are stored in the underlying distributed file system [23].

The advantage of using MapReduce is that the programmer doesn't need to get involved in low-level details and complexities of issues like data distribution, scheduling and fault tolerance. However, MapReduce has some limitations regarding distributed processing of large scale graphs:

- MapReduce is not inherently designed for processing graph data. Instead it is tailor-made for data represented as key-value pairs. This introduces significant complexity and overhead for porting graph algorithms to this programming model [27].
- While most graph algorithms are iterative and therefore, they should be implemented by multiple iterations of map and reduce stages. However, MapReduce does not intrinsically support data communication between successive iterations of map and reduce. In other words, after each iteration the results are stored on the underlying file system, and they should be reloaded in the next iteration. The I/O overhead due to frequent data serialization might lead to considerable inefficiency in case of large graphs [6][10][21][27].
- The map and reduce workers are located in different physical machines. As a result, a constant network connection is required which is very disk-intensive [10].
- Many graph algorithms are exploratory in the sense the graph is browsed from vertex to vertex during the execution of the algorithm. Because of the poor-locality challenge, it is frequently needed to access data stored on a remote machine. This may lead to significant overhead if MapReduce is used, since it does not fundamentally support direct communication between processing nodes [10][21][27].

Next, some of the works which use MapReduce programming model for large scale graph processing are described.

PEGASUS [17] is the first open source library for large scale graph processing based on MapReduce. It is based on the idea that most graph algorithms can be translated to vector-matrix multiplication problem. Therefore, it uses an adjacency matrix representation format for the graph data, and divides the corresponding multiplication into three main operations which their specific details are specified by the programmer. PEGASUS then uses MapReduce to execute each of the three operations. It also provides some improvements like dividing the corresponding matrix and vector into equal blocks.

MGMF [20] is a MapReduce based framework for large scale graph processing. It defines four categories of graph mining algorithms based on how data of a graph is accessed and collected: Traverse all, Traverse Partial, One-Hop, and Multi-Hop. Further, it proposes a framework for porting algorithms of each category to MapReduce model. This framework consists of three main components, i.e. primitive functions, distributed algorithms, and optimization methods. It is implemented as an open-source library and experimental results demonstrate that it outperforms PEGASUS in terms of scalability and performance.

Surfer [18] is another large scale graph processing engine. Its goal is to provide better efficiency and more programmability by supporting two programming models, i.e. MapReduce and Propagation, for implementing graph algorithms. In order to address the challenges of MapReduce for graph processing, Surfer complements MapReduce with the two steps of Propagation programming model, i.e. transfer and combine. Surfer uses Propagation model to improve data communication between processing machines. It introduces a bandwidth-aware algorithm for partitioning the input graph so that communication overhead is reduced. Further it enables executing user-defined functions over the partitioned graph in a distributed manner.

GBASE [19] proposes a framework that consists of two processes, i.e. indexing and query process. In the former, the input graph which is represented as an adjacency matrix is partitioned into several blocks. Then, GBASE reshuffles the adjacency matrix in order to put the vertices that are in the same partition near each other. Finally, all non-empty blocks are compressed by a standard compression method and stored in the distributed file system. In the query process, GBASE unifies different types of operations through matrix-vector multiplication and selects the required files before executing the query as MapReduce jobs.

C. *Vertex-Centric BSP-based*

Vertex-centric is a programming model in which computations of a graph algorithm take place in each vertex. In other words, the algorithm is implemented by considering each vertex as a computational unit that has information about itself and its outgoing edges. The advantage of this programming model is its simplicity and its suitability for many graph algorithms [6]. On the other hand, it has some efficiency issues since each vertex has a very limited knowledge about the graph, and for instance it might have to spend many iterations for propagating its data to another vertex, even if both vertices are stored on a same graph partition on the same machine. Recently, graph-centric programming is introduced to address this issue by considering larger computational units which have access to the whole partition structure. So it enables utilization of the local graph structure in a partition, and consequently supports more complex and flexible graph algorithms [29].

BSP (Bulk Synchronous Parallel) is a computational model for parallel processing. It performs the computations in a sequence of supersteps where each superstep consists of three minor steps [24]:

1. Parallel computational step in which different processes run simultaneously,

2. Communication step in which different processes send message to each other to communicate data,
3. Synchronization step in which each process waits until all the processes have reached to a specific state, ready for starting the next superstep.

In comparison with the MapReduce model, the BSP model has the advantage that it is stateful and doesn't need the graph structure to be transferred between the processing machines. Therefore, it has less communication overhead. Further, BSP model can utilize in-memory computation to improve efficiency if the whole graph structure fits in the memory of all machines [21]. Otherwise, it is required to implement spilling-to-disk techniques. As a result, MapReduce is still the best alternative for parallel processing of large-scale graphs when the graph structure doesn't fit in the local memory [10].

In vertex-centric BSP programming model, each vertex of the graph is a computational unit which executes BSP supersteps [6]. In the following, some of the works which are based on this programming model are described.

In 2010, Google introduced Pregel [15] as a scalable framework for large-scale graph processing. Pregel executes a graph algorithm as a set of iterations in different supersteps. During each superstep *S*, Pregel calls, in parallel, a user-defined function for each vertex V. This function can 1) read the messages that vertex V has received in the previous superstep, 2) change the state of vertex V or its associated outgoing edges, and 3) send messages to other vertices, so that these messages are received at the destination vertex in the next superstep. Generally speaking, each superstep is a unit of parallel computation. There is a synchronization phase between any two successive supersteps which is responsible for checking that every vertex receives the messages of the previous superstep. The algorithm is terminated when there are no messages to be received and all vertices have become inactive. The underlying idea of Pregel has motivated many researchers to develop an open source version of Pregel. Phoebus[3], GoldenOrb[4] and Giraph[5] are some examples.

Apache Giraph is the most popular open-source implementation of Pregel. It uses Hadoop [22] and Zookeeper frameworks, and is based on a master/slave architecture. The master node is responsible for partitioning the graph before each superstep. It also distributes the graph vertices between worker nodes and monitors state of these nodes during the superstep. Barrier synchronization at each superstep is also executed by master node. On the other hand, each worker node executes the user defined function on each vertex assigned to it, and if it is necessary, it sends messages to other worker nodes. The role of ZooKeeper is to do coordination management at the application level, and provide fault tolerance.

GPS [16] is another open-source project for large-scale graph processing that is inspired by Pregel [15]. However, it has three main differences with Pregel framework or with Giraph. In addition to vertex-centric algorithms, GPS allows combining several vertex-centric computations with global computations. GPS utilizes repartitioning during the computations in order to reduce communication overhead. This is performed by moving to a single machine the vertices which frequently send messages to each other. Finally, while Pregel executes the computations of the vertices in parallel, but it does not support parallelization of the computation of a single vertex. This might reduce utilization when a vertex has heavy computation due to its high degree. GPS addresses this problem by distributing the adjacency list of high degree vertices over multiple machines to enable parallel processing for each vertex.

Another similar system is DisNet [11] which has a master/slave architecture in which every worker node is responsible for performing computations over a subset of the vertices, collecting the local results and sending them to the master node. DisNet is similar to Pregel since it follows the vertex-centric and synchronous programming model. However, it has two main differences with Pregel: 1) in Disnet, workers are allowed to communicate only with the master node, while in Pregel worker nodes can communicate with each other. 2) Pregel partitions the input graph over the workers, but DisNet stores a copy of the entire graph in the memory of each worker, and therefore, workers need less communications for graph traversal. This increases the performance, but also limits the scalability in case of very large graphs that do not fit in the main memory.

Seraph [30] is a low-cost system for large-scale graph processing. The goal of designing Seraph is to improve inefficient use of memory and high cost of fault tolerance in case of running multiple concurrent jobs which is the drawback of most of the existing systems. To achieve this goal, it allows concurrent jobs to share graph structure in local memory and proposes a new incremental checkpoint model. Its computational model is derived from the Pregel. However it separates its data model and computing logics. Evaluation results demonstrate that Seraph significantly outperforms Giraph in terms of execution time and memory utilization.

Trinity [31] is a general-purpose graph processing engine which works on cloud distributed memory. It supports efficient graph processing through optimization of memory management and network communication mechanisms. This is achieved by differentiating between two types of graph access patterns, i.e. online and offline access pattern, and

[3] Phoebus. https://github.com/xslogic/phoebus

[4] GoldenOrb. http://www.raveldata.com/goldenorb

[5] Giraph. http://giraph.apache.org

providing customized mechanisms for each type. It must be noted that other works, except for graph databases, support only offline pattern.

In case of online access pattern, which is used for query execution on graphs, Trinity employs key/value storage model where structure of the graph is kept in a distributed in-memory store. On the other hand, for offline access pattern which is used in graph analytics tasks, Trinity uses vertex-centric programming model. While this makes it similar to Pregel, Trinity is different in the sense that unlike Pregel which allows a vertex to send and receive messages to and from any other vertex, Trinity restricts a vertex to send and receive messages to and from only a fixed set of vertices. This restriction enables better graph partitioning and consequently, optimization of message passing schema. Another difference is that while Pregel is based on synchronous BSP model, but Trinity is more flexible and supports asynchronous model in addition to synchronous BSP. Experimental evaluations demonstrate that Trinity outperforms BPGL [9] and Giraph in terms of execution time and memory utilization.

D. Vertex-Centric Asynchronous-based

Despite its benefits, the BSP model has the disadvantage that it might increase the cost of convergence between computations of different vertices. In other words, the barrier synchronization introduces some latency, as faster workers should wait for slower ones. Asynchronous computations can address this issue by accelerating this convergence through mechanisms for faster propagation of the vertex updates, since updates of a vertex can be used by the next vertices as soon as they become available, and it is not needed to wait until the message passing step [6][14][28].

While asynchronous parallel programming model can lead to much better efficiency compared to synchronous model, but its main problem is its programming complexity. From a practical point of view, it is very hard to implement, test and specially debug an asynchronous parallel program. Also there are some frameworks which seek to address this problem by providing new abstractions in asynchronous programming, but they are still far from the simplicity of programming in synchronous model. For instance, they require the user to get involved in low level concurrency issues which is not negligible requirement [6][28].

Next, some works which are based on vertex-centric asynchronous programming model are described.

GraphLab [14] uses asynchronous distributed shared memory architecture, in which each vertex program, in addition to the information of the vertex, can directly access the information of its adjacent vertices and edges (direction of edges is not important). In other words, each machine which is responsible for a graph partition stores information of remote adjacent vertices and edges as ghost cells. Further vertex programs can schedule their neighboring vertex programs for running in the future. In GraphLab, computation is organized as a number of tasks, where every task is responsible for executing the update function of a vertex, its edges and its adjacent vertices. Further, GraphLab provides different scheduling algorithms for ordering updates of the tasks.

GraphLab eliminates the need to communicate messages to transfer the information over network. Consequently, it isolates user defined algorithms from the data communications and allows the underlying system to choose when and how to move program state.

PowerGraph [25] seeks to eliminate challenges of Pregel [15] and GraphLab [14] for processing large graphs in the real world i.e. natural graphs. The main property of natural graphs is that the majority of vertices have a few neighbors, while a minority of them have many neighbors. This happens for instance in social networks where celebrities are connected to a much greater number of users than the normal users. This property makes it hard to partition the graph in such a way that the network communications is minimized.

Pregel and GraphLab employ hash functions for partitioning, and this leads to unbalance workload distribution when applied to natural graphs. In these systems, machines which are responsible for high degree vertices need a large amount of memory to store the information of the vertices. In addition, the execution of each vertex program which is expensive for high-degree vertices cannot be parallelized. These characteristics lead to limited performance and scalability in these systems.

PowerGraph uses the simple idea of vertex centric programming. However, it enables the information of high degree vertices to be distributed over all the machines. It utilizes the distributed shared memory technique and the associative gather concept correspondingly from GraphLab and Pregel, and can execute vertex programs both in synchronous and asynchronous mode. Experimental results demonstrate that PowerGraph outperforms GraphLab and Pregel in terms of performance and scalability.

GRACE [28] is another asynchronous system, which allows the user to implement his graph algorithms using BSP model which is synchronous, and therefore advantages of BSP, i.e. easier implementation, testing and debugging. In other words, it supports using asynchronous execution policies in BSP, which improves convergence rate of the algorithms. This advantage is provided through relaxation of the two constrains of the BSP model, i.e. isolation and consistency. Isolation means that in each BSP superstep, the messages sent by a vertex are not received by the target vertices until the next superstep. GRACE relaxes this constraint by allowing messages of a vertex to be visible to other vertices in the same superstep. This way, by careful ordering of update function execution on the vertices, it is possible to

make considerable improvements. For instance, the vertices with higher priority messages can be processed earlier so that their messages affect computations of the other vertices.

Consistency in BSP means that a vertex program is executed if and only if all the input messages of that vertex are available at the beginning of the superstep. GRACE relaxes this constraint by allowing a vertex program to be started before it has received all its input messages, for instance, as soon as it has received at least one message. GRAE provides facilities for the programmer to specify his definition of the relaxed constraints. Experimental evaluations show that asynchronous execution of BSP in GRACE can reach convergence rate of fully asynchronous models like GraphLab. It also preserves near-linear scalability of synchronous BSP models by minimizing concurrency control overhead.

HIPG [12] is another vertex-centric asynchronous graph processing model which uses principles of object oriented programming. Each vertex is an object which contains its values and outgoing edges. Further, each vertex program has access to other vertex objects through a specific function. This function is identical for accessing both local and remote vertices. When the function is called for a remote vertex, HIPG replaces it with an asynchronous message sent to the target machine. The main advantage of HIPG is that it provides a high-level programming interface which hides some complexities, e.g. explicit message passing, from the programmer and provides more flexibility, e.g. by allowing definition of custom attributes and methods for the vertices. Further, it provides good facilities for using divide-and-conquer technique which is very useful in graph algorithms.

IV. CONCLUSION

In this paper different works on distributed large-scale graph processing were reviewed. These works are categorized into four groups based on their computational model. The MPI-based models have high performance but their programming model is not easy and the programmer is involved in low level complexities like message passing, concurrency control and fault tolerance.

Models based on Map-Reduce have the advantage that they free the programmer from dealing with much details like data distribution, fault tolerance and replication management. However, since Map-Reduce is not inherently designed for graph processing, the iterative nature of most graph algorithms increases I/O overhead of using Map-Reduce.

The advantage of vertex-centric BSP models is programming simplicity of vertex-centric model and also high performance of BSP which makes it an appropriate model for graph algorithms which include mostly iterative computations. Since BSP stores the graph structure in the memory of each machine, it eliminates the need to transfer this structure between machines. This reduces network communication overhead and improves execution performance. The disadvantage of BSP based models is the low convergence rate due to the synchronization overhead.

Asynchronous programming models address this challenge, however in the cost of a more complex programming model, which increases difficulty of implementing, testing and debugging distributed graph algorithms.

Vertex-Centric and Map-Reduce based systems are more abstract and convenient for implementation in comparison with MPI based systems, however their API for large scale graph processing are low-level and they use non-traditional programming models to maximize parallelism and scalability. Further, they don't offer mechanisms for controlling the flow of programs. As a result, it leads to large and complex programs for writing sophisticated graph algorithms like those composed of multiple operations and needs global control flows like Betweenness Centrality computation algorithm.

Some recent works have tried to address this problem. For instance, HelP [33] is a set of high-level graph processing primitives that abstract the most commonly appearing operations in distributed graph computations. As another example, [34] has designed a compiler to translate imperative graph algorithms into equivalent Pregel implementations using semantics defined in Green-Marl as a Domain-Specific Language for graph analysis.

The works reviewed in this paper are summarized in Table 1, in terms of their main properties. Some of the concluding points which are also shown in this table are:

- Using vertex-centric programming model is the dominant trend in the recent years. This can be attributed to the simplicity of thinking and designing graph algorithms based on this model.
- The majority of proposed systems use adjacency list for graph representation, instead of adjacency matrix. One reason is that in the matrix-based representation, the computations of the graph algorithm might need to be translated into vector-matrix multiplication which consequently increases programming difficulty.
- Most proposed models incorporate fault-tolerance. This is not unexpected since fault tolerance is an importance quality attribute in distributed systems.
- Most works employ some graph partitioning mechanism for the purpose of data distribution among the processing machines. This improves scalability and enables processing larger graphs.
- There are a few numbers of vertex-centric models that support parallelization of the vertex program. This improves performance,

especially when dealing with highly unstructured real worlds graphs, because heavy computation in high-degree vertices is improved through parallelization.

The main conclusion achieved through studying existing works is that, each group of the proposed models has interesting advantages, which cannot be ignored in comparison with other groups. Therefore, we consider this as a main research direction to identify key characteristics and requirements of different graph algorithms, and to analyze each of the proposed programming models to determine how naturally they cope with those characteristics and how effectively they address the corresponding requirements. This way it is possible to propose efficient and tailor-made models for each category of graph algorithms.

The distinction made by Trinity [31] between two categories of graph access patterns is an example of the categorization schema suggested by this paper. However, our point of view is that there is much work needed to enhance this schema by extending its coverage.

REFERENCES

[1] M. Lambertini, M. Magnani, M. Marzolla, D. Montesi, and C. Paolino, "Large Scale Social Network Analysis", Large Scale Data Analytics, pp. 155-187, 2014.

[2] J. Scott, "Social Network Analysis: A Handbook", London, SAGE Publications, 2000.

[3] K.K.S. Chung, L. Hossain, and J. Davis, "Exploring Sociocentric and Egocentric Approaches for Social Network Analysis", International Conference on Knowledge Management, pp. 1–8, 2005.

[4] I. Robinson, J. Webber, and E. Eifrem, "Graph databases", O'Reilly Media publisher, 2013.

[5] S. Sakr, "Processing large-scale graph data: A guide to current technology", IBM developerWorks Article, June 2013.

[6] M.U. Nisar, A. Fard, and J.A. Miller, "Techniques for Graph Analytics on Big Data", IEEE Big Data, pp. 255- 262, 2013.

[7] A. Lumsdaine, D. Gregor, B. Hendrickson, and J.W. Berry, "Challenges in parallel graph processing", Parallel Process. Lett. Vol. 17, No.1, pp. 5–20, 2007.

[8] Message Passing Interface Forum. MPI: A message passing interface. In Proc. Supercomputing '93, pp. 878-883, 1993.

[9] D. Gregor and A. Lumsdaine, "The Parallel BGL: A generic library for distributed graph computations", Parallel Object-Oriented Scientific Computing, pp.1-18 , 2005.

[10] T. Kajdanowicz, P. Kazienko, W. Indyk, "Parallel processing of large graphs", Future Generation Computer Systems, Vol. 32, pp. 324-337, 2014.

[11] R.N. Lichtenwalter and N.V. Chawla, "DisNet: A framework for distributed graph computation", International Conference on Social Networks Analysis and Mining (ASONAM), pp. 263-270, 2011.

[12] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal, "A high-level framework for distributed processing of large-scale graphs", Distributed Computing and Networking, pp. 155-166, 2011.

[13] A. Buluc, and J.R. Gilbert, "The Combinatorial BLAS: design, implementation, and applications", International Journal of High Performance Computing Applications, Vol.25, No.4 , pp. 496–509 , 2011.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud", PVLDB, 2012 .

[15] G. Malewicz, M.H. Austern, A.J. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing", SIGMOD, pp. 135-146 , 2010.

[16] S. Salihoglu and J. Widom, "GPS: A Graph Processing System", Scientific and Statistical Database Management, 2013.

[17] U. Kang, C.E. Tsourakakis, and C. Faloutsos, "PEGASUS: mining peta-scale graphs", Knowledge Information System, Vol.27, No.2, pp.303–325, 2011.

[18] R. Chen, X. Weng, B. He, M. Yang, B. Choi, and X. Li, "On the Efficiency and Programmability of Large Graph Processing in the Cloud", Microsoft Research TechReport, 2010.

[19] U. Kang, H. Tong, J. Sun, C.Y. Lin, and C. Faloutsos, "Gbase: an efficient analysis platform for large graphs", VLDB Journal., Vol.21 No.5, pp.637-650, 2012.

[20] Y-C. Lo, H-C. Lai, C-T. Li, and S-D. Lin, "Mining and generating large-scaled social networks via MapReduce", Social Network Analysis and Mining, Vol.3, No.4, pp. 1449-1469 , 2013.

[21] T. Kajdanowicz, W. Indyk, P. Kazienko, and J. Kukul, "Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing", ICDMW, pp. 218-225, 2012.

[22] T. White, "Hadoop: The Definitive Guide", Ed. O'Reilly, 2010.

[23] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", Sixth Symposium on Operating System Design and Implementation, pp. 10-17, 2004.

[24] L.G. Valiant, "A bridging model for parallel computation", Communications of the ACM, Vol. 33, No. 8, pp. 103–111, 1990.

[25] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), pp.17-30, 2012.

[26] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. LI, "Improving Large Graph Processing on Partitioned Graphs in the Cloud", 3rd ACM Symposium on Cloud Computing (SoCC '12), pp. 1–13, 2012.

[27] D. CORDEIRO, A. GOLDMAN, A. KRAEMER, and F.P. JUNIOR, "Using the BSP model on Clouds", Cloud Computing and Big Data, pp. 123-141, 2013.

[28] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy", Biennal Conference on Innovative Data Systems Research (CIDR '13), 2013.

[29] Y. Tian, A. Balmin, S.A. Corsten, S. Tatikonday, and J. McPherson, "From 'Think Like a Vertex' to 'Think Like a Graph' ", VLDB Endowment, Vol. 7, No. 3, 2013.

[30] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an Efficient, Low-cost System for Concurrent Graph Processing", 23rd international symposium on High-performance parallel and distributed computing (HPDC '14), pp. 227-238, 2014.

[31] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud", ACM International Conference on Management of Data (SIGMOD '13), pp. 505–516, 2013.

[32] L-Y. Ho, J-J. Wu, and P. Liu, "Data Replication for Distributed Graph Processing", Sixth International Conference on Cloud Computing , pp. 319-326, 2013.

[33] S. Salihoglu, J. Widom, "HelP: High-level Primitives For Large-Scale Graph Processing", Workshop on GRAph Data management Experiences and Systems (GRADES'14), pp. 1-6, 2014.

[34] S. Hong, S. Salihoglu, J. Widom, K. Olukotun, "Simplifying Scalable Graph Processing with a Domain-Specific Language", Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2014.

Table 1. Summary of distributed graph processing systems

System	Computational Model	Programming language	Open Source	Matrix-Based model	Fault-Tolerance	Data Distribution method	parallel Vertex Program
BPGL [9]	MPI	C++	*			hybrid	
BALS [13]	MPI	C++		*		partitioning	
PEGASUS [17]	MapReduce	java	*	*	*	partitioning	
MGMF [20]	MapReduce	C++	*		*	partitioning	
Surfer [18]	MapReduce	C++			*	partitioning	
GBASE [19]	MapReduce	?		*	*	partitioning	
Pregel [15]	Vertex-Centric BSP	C++			*	partitioning	
Giraph	Vertex-Centric BSP	java	*		*	partitioning	
GPS [16]	Vertex-Centric BSP	java	*		*	partitioning	*
Seraph [30]	Vertex-Centric BSP	java			*	partitioning	
DiNet [11]	Vertex-Centric BSP	C++				copy	
Trinity [31]	Vertex-Centric Asynchronous/synchronous	C++			*	partitioning	
GraphLab [14]	Vertex-Centric Asynchronous	C++	*		*	hybrid	
PowerGraph [25]	Vertex-Centric Asynchronous/synchronous	C++			*	hybrid	*
GRACE [28]	Vertex-Centric Asynchronous	C++				partitioning	
HipG [12]	Vertex-Centric Asynchronous	java				partitioning	

Mohsen Kahani is currently a Full Professor and Chief of Information Office (CIO) of Ferdowsi University of Mashhad, Iran. He received his BS in 1990, from the University of Tehran, Iran, his M.Sc. in 1994, and his Ph.D. in 1998 both from University of Wollongong, Australia. His research interests include information technology, software engineering, semantic web and linked data.

Saeid Abrishami received his M.Sc. and Ph.D. degrees in computer engineering from Ferdowsi University of Mashhad, in 1999 and 2011, respectively. Since 2003, he has been with the Department of Computer Engineering of Ferdowsi University of Mashhad, where he is currently an assistant professor. During the spring and summer of 2009 he was a visiting researcher at the Delft University of Technology. His research interests focus on resource management and scheduling in the distributed systems, especially in Grids and Clouds.

Fattane Zarrinkalam is currently doing her PhD degree in Software Engineering at Ferdowsi University of Mashhad, Iran. She received her B.Sc. and M.Sc. degrees in computer engineering from Ferdowsi University of Mashhad, in 2009 and 2011, respectively. Her research interests include social network analysis, semantic web and linked data.

This Page intentionally left blank.