

Big Graph Analytics Platforms

Da Yan

The University of Alabama at Birmingham
yanda@uab.edu

Yingyi Bu

Couchbase, Inc.
yingyi@couchbase.com

Yuanyuan Tian

IBM Almaden Research Center, USA
ytian@us.ibm.com

Amol Deshpande

University of Maryland
amol@cs.umd.edu

now

the essence of knowledge

Boston — Delft



Documents

Document Sharing Hub

Foundations and Trends[®] in Databases

Published, sold and distributed by:

now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:

now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

D. Yan, Y. Bu, Y. Tian, and A. Deshpande. *Big Graph Analytics Platforms*.
Foundations and Trends[®] in Databases, vol. 7, no. 1-2, pp. 1–195, 2015.

This Foundations and Trends[®] issue was typeset in L^AT_EX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-242-6

© 2017 D. Yan, Y. Bu, Y. Tian, and A. Deshpande

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without prior written permission of the publishers.

Photocopying. In the USA: This journal is registered at the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923. Authorization to photocopy items for internal or personal use, or the internal or personal use of specific clients, is granted by now Publishers Inc for users registered with the Copyright Clearance Center (CCC). The 'services' for users can be found on the internet at: www.copyright.com

For those organizations that have been granted a photocopy license, a separate system of payment has been arranged. Authorization does not extend to other kinds of copying, such as that for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. In the rest of the world: Permission to photocopy must be obtained from the copyright owner. Please apply to now Publishers Inc., PO Box 1024, Hanover, MA 02339, USA; Tel. +1 781 871 0245; www.nowpublishers.com; sales@nowpublishers.com

now Publishers Inc. has an exclusive license to publish this material worldwide. Permission to use this content must be obtained from the copyright license holder. Please apply to now Publishers, PO Box 179, 2600 AD Delft, The Netherlands, www.nowpublishers.com; e-mail: sales@nowpublishers.com

Foundations and Trends[®] in Databases

Volume 7, Issue 1-2, 2015

Editorial Board

Editor-in-Chief

Joseph M. Hellerstein

University of California, Berkeley
United States

Editors

Anastasia Ailamaki
EPFL

Peter Bailis
University of California, Berkeley

Mike Cafarella
University of Michigan

Michael Carey
University of California, Irvine

Surajit Chaudhuri
Microsoft Research

Minos Garofalakis
Yahoo! Research

Ihab Ilyas
University of Waterloo

Christopher Olston
Yahoo! Research

Jignesh Patel
University of Michigan

Chris Re
Stanford University

Gerhard Weikum
Max Planck Institute Saarbrücken



Documents
Document Sharing Hub

Editorial Scope

Topics

Foundations and Trends® in Databases covers a breadth of topics relating to the management of large volumes of data. The journal targets the full scope of issues in data management, from theoretical foundations, to languages and modeling, to algorithms, system architecture, and applications. The list of topics below illustrates some of the intended coverage, though it is by no means exhaustive:

- Data models and query languages
- Query processing and optimization
- Storage, access methods, and indexing
- Transaction management, concurrency control, and recovery
- Deductive databases
- Parallel and distributed database systems
- Database design and tuning
- Metadata management
- Object management
- Trigger processing and active databases
- Data mining and OLAP
- Approximate and interactive query processing
- Data warehousing
- Adaptive query processing
- Data stream management
- Search and query integration
- XML and semi-structured data
- Web services and middleware
- Data integration and exchange
- Private and secure data management
- Peer-to-peer, sensornet, and mobile data management
- Scientific and spatial data management
- Data brokering and publish/subscribe
- Data cleaning and information extraction
- Probabilistic data management

Information for Librarians

Foundations and Trends® in Databases, 2015, Volume 7, 4 issues. ISSN paper version 1931-7883. ISSN online version 1931-7891. Also available as a combined paper and online subscription.

Big Graph Analytics Platforms

Da Yan

The University of Alabama at Birmingham
yanda@uab.edu

Yingyi Bu

Couchbase, Inc.
yingyi@couchbase.com

Yuanyuan Tian

IBM Almaden Research Center, USA
ytian@us.ibm.com

Amol Deshpande

University of Maryland
amol@cs.umd.edu

Contents

1	Introduction	2
1.1	History of Big Graph Systems Research	3
1.2	Features of Big Graph Systems	6
1.3	Organization of the Survey	13
2	Preliminaries	17
2.1	Data Models and Analytics Tasks	17
2.2	Distributed Architecture	19
2.3	Single-Machine Architecture	23
I	Vertex-Centric Programming Model	25
3	Vertex-Centric Message Passing (Pregel-like) Systems	26
3.1	The Framework of Pregel	26
3.2	Algorithm Design in Pregel	29
3.3	Optimizations in Communication Mechanism	34
3.4	Load Balancing	37
3.5	Out-Of-Core Execution	40
3.6	Fault Tolerance	44
3.7	Summary	50

4	Vertex-Centric Message-Passing Systems Beyond Pregel	51
4.1	Block-Centric Computation	51
4.2	Asynchronous Execution	63
4.3	Vertex-Centric Query Processing	69
4.4	Summary	72
5	Vertex-Centric Systems with Shared Memory Abstraction	73
5.1	Distributed Systems with Shared Memory Abstraction . . .	74
5.2	Out-of-Core Systems for a Single PC	80
5.3	Summary	90
II	Beyond Vertex-Centric Programming Model	92
6	Matrix Algebra-Based Systems	93
6.1	PEGASUS	93
6.2	GBASE	95
6.3	SystemML	97
6.4	Summary	99
7	Subgraph-Centric Programming Models	103
7.1	Complex Analysis Tasks	104
7.2	NScale	109
7.3	Arabesque	110
7.4	Summary	112
8	DBMS-Inspired Systems	113
8.1	The Recursive Query Abstraction	115
8.2	Dataflow-Based Graph Analytical Systems	121
8.3	Incremental Graph Processing	129
8.4	Integrated Analytical Pipelines	131
8.5	Summary	134
III	Miscellaneous Issues	135
9	More on Single-Machine Systems	136

- 9.1 Vertex-Centric Systems with Matrix Backends 136
- 9.2 In-Memory Systems for Multi-Core Execution 142
- 9.3 Summary 148
- 10 Hardware-Accelerated Systems 150
 - 10.1 Out-of-Core SSD-Based Systems 150
 - 10.2 Systems for Execution with GPU(s) 154
 - 10.3 Summary 159
- 11 Temporal and Streaming Graph Analytics 161
 - 11.1 Overview 162
 - 11.2 Historical Graph Systems 164
 - 11.3 Streaming Graph Systems 170
 - 11.4 Brief Summary of Other Work 174
 - 11.5 Summary 176
- 12 Conclusions and Future Directions 178
- References 182

Abstract

Due to the growing need to process large graph and network datasets created by modern applications, recent years have witnessed a surging interest in developing big graph platforms. Tens of such big graph systems have already been developed, but there lacks a systematic categorization and comparison of these systems. This article provides a timely and comprehensive survey of existing big graph systems, and summarizes their key ideas and technical contributions from various aspects. In addition to the popular vertex-centric systems which espouse a think-like-a-vertex paradigm for developing parallel graph applications, this survey also covers other programming and computation models, contrasts those against each other, and provides a vision for the future research on big graph analytics platforms. This survey aims to help readers get a systematic picture of the landscape of recent big graph systems, focusing not just on the systems themselves, but also on the key innovations and design philosophies underlying them.

D. Yan, Y. Bu, Y. Tian, and A. Deshpande. *Big Graph Analytics Platforms*. Foundations and Trends[®] in Databases, vol. 7, no. 1-2, pp. 1–195, 2015.
DOI: 10.1561/19000000056.

1

Introduction

The growing need to deal with massive graphs in real-life applications has led to a surge in the development of big graph analytics platforms. Tens of big graph systems have already been developed, and more are expected to emerge in the near future. Researchers new to this young field can easily get overwhelmed and lost by the large amount of literature. Although several experimental studies have been conducted in recent years that compare the performance of several big graph systems [Lu et al., 2014, Han et al., 2014a, Satish et al., 2014, Guo et al., 2014], there lacks a comprehensive survey that clearly summarizes the key features and techniques developed in existing big graph systems. A recent survey [McCune et al., 2015] attempts to cover the landscape as well, but primarily focuses on vertex-centric systems; it omits most of the work on other programming models and also several crucial optimization and programmability issues with vertex-centric systems. In addition to describing the various systems, this survey puts more emphasis on the innovations and technical contributions of existing systems, in order to help readers quickly obtain a systemic view of the key ideas and concepts. We hope this will help big graph system researchers

avoid reinventing the wheel, apply useful existing techniques to their own systems, and come up with new innovations.

In the rest of this chapter, we first review the history of research on Big Graph systems, and then overview some important features of existing Big Graph systems. Finally, we present the organization of this survey. Many contents of this survey are covered by our tutorial in SIGMOD 2016 [Yan et al., 2016a], the slides of which are available online¹ and contain animations to illustrate the various techniques used by existing systems.

1.1 History of Big Graph Systems Research

Although graph analytics has always been an important research topic throughout the history of computation, the research on *big* graph processing only flourished in recent years as part of the big data movement, which has seen increased use of advanced analytics on large volumes of unstructured or semi-structured data. A hallmark of this movement has been the MapReduce distributed data processing framework, introduced by Google [Dean and Ghemawat, 2004], and the companion Google File System (GFS) [Ghemawat et al., 2003]. Subsequently, the Apache Hadoop project² implemented the open-source counterparts, the Hadoop Distributed File System (HDFS) and the Hadoop MapReduce framework in 2006. Since then, a huge body of research has focused on designing novel MapReduce algorithms as well as on improving the framework for particular workloads. A large body of work in that space focused on big graph analytics, and many tailor-made MapReduce algorithms were proposed for solving specific graph problems [Lin and Schatz, 2010]. As an early MapReduce-based framework designed for general-purpose graph processing, PEGASUS [Kang et al., 2009] models graph computation by a generalization of matrix-vector multiplication. However, the reliance on the disk-based Hadoop MapReduce runtime, which requires repeated reads and writes of large files from HDFS, fundamentally limits its performance.

¹http://www.cse.cuhk.edu.hk/systems/gsys_tutorial/

²<https://hadoop.apache.org/>

Later, [Malewicz et al. \[2010\]](#) proposed the Pregel framework specially designed for large-scale big graph processing. Since many graph algorithms are iterative, Pregel keeps the graph data in the main memory and adopts an iterative, message-passing computation model (inspired by the well-known Bulk Synchronous Parallel model for parallel computation), and is thus much more efficient than MapReduce. Pregel also adopts a “think-like-a-vertex” programming model which is more intuitive and user-friendly for average programmers and a natural fit for a range of graph analysis tasks. The vertex-centric programming model of Pregel is also very expressive since a vertex can communicate with any other vertex by passing messages. Since the introduction of Pregel, it has sparked a large number of research works on extending the basic Pregel framework in different aspects to improve the graph processing performance [[Tian et al., 2013](#), [Yan et al., 2014a](#), [Zhang et al., 2014](#), [Han and Daudjee, 2015](#), [Yan et al., 2016b](#)].

Independent of Pregel, [Low et al. \[2010\]](#) developed a multi-core, shared-memory graph-based computation model, called GraphLab. Then, [Low et al. \[2012\]](#) extended it to work in a distributed environment, while keeping the shared memory programming abstraction, in which a vertex can directly access the states of its adjacent vertices and edges. Later, GraphLab switched to the GAS (Gather-Apply-Scatter) computation model to further improve the system performance [[Gonzalez et al., 2012](#)]. Although the GAS model covers a large number of graph algorithms, it is less expressive than the Pregel model, since a vertex can only access the data of its adjacent vertices and edges; we call this a *neighborhood-based shared memory* abstraction. This programming abstraction is especially popular among recent big graph systems designed to run on a single machine, such as GraphChi [[Kyrola et al., 2012](#)].

While Pregel and GraphLab are designed specially for graph processing, a number of systems, such as GraphX [[Gonzalez et al., 2014](#)] and Pregelix [[Bu et al., 2014](#)], rely on a general-purpose data processing engine for execution, at the same time providing graph-specific programming interfaces similar to those in Pregel and GraphLab.

Vertex-centric systems are ideally suited for graph analysis tasks like PageRank computation where the overall computation can be broken down into individual tasks, each involving a specific vertex (i.e., its local state, and the states of its adjacent edges). Many machine learning tasks (e.g., belief propagation, matrix factorization, stochastic gradient descent) are also a natural fit for those systems. However, many complex graph analysis tasks cannot be easily decomposed in such fashion. For example, a class of graph problems termed “ego-centric analysis” [Quamar et al., 2016] require analyzing the neighborhoods of the vertices in their entirety. Also, graph problems such as graph matching or graph mining may have intermediate or output results with size superlinear or even exponential in the input graph size. Complex graph algorithms, e.g., the Hungarian algorithm for maximum bipartite matching, even require random access to the entire graph. Solving these problems using vertex-centric processing leads to substantial communication and memory overheads, since each vertex needs to collect the relevant neighborhood subgraph (if not the entire graph) to its local state before processing the subgraph.

This has led to the development of many alternative programming frameworks, examples of which include Socialite [Seo et al., 2013b], Arabesque [Teixeira et al., 2015], NScale [Quamar et al., 2016], among others. In addition, several systems including Ligra [Shun and Blelloch, 2013], Galois [Nguyen et al., 2013], Green-Marl DSL [Hong et al., 2012], etc., provide low-level graph programming frameworks that can handle nearly arbitrary graph computations. These frameworks often focus on specific classes of graph problems, and make a range of different assumptions about the computing environment, making them incomparable in many cases. Arabesque tackles problems like graph matching and graph mining where the intermediate result can be very large, while assuming that the entire graph can be held in a single machine memory. NScale is a strict generalization of the vertex-centric framework, and can handle tasks that require access to multi-hop neighborhoods of vertices; but it does not support the other classes of problems discussed above. Socialite uses a Datalog-inspired programming model which is most suitable for graph problems that can be expressed as recursive

Datalog queries. Ligra, Galois, and other similar systems require random access to the graph and focus on large-memory multi-core environments. Thus, developing a sufficiently expressive, yet easy-to-use and easy-to-parallelize graph programming model, remains a critical and open challenge in this field.

The majority of existing big graph systems are designed for processing static graphs (or with small topology mutations). However, real-world graphs often evolve over time, with vertices and edges continually being added or deleted, and their attributes being frequently updated. A new class of big graph systems, such as KineoGraph [Cheng et al., 2012], TIDE [Xie et al., 2015b], DeltaGraph [Khurana and Deshpande, 2013], and Chronos [Han et al., 2014b], have emerged to process and analyze temporal and streaming graph data. This area is however still in its infancy and there are many open problems that need to be addressed to effectively handle continuous and/or temporal analytics on big graphs.

There is also a large body of work on executing queries related to a specific vertex (or a small subset of vertices) against large volumes of graph data, which has developed a range of specialized indexes and search algorithms. This survey does not cover that body of work.

1.2 Features of Big Graph Systems

We can categorize the big graph platforms along various dimensions. Since an important feature of the modern big graph systems is user-friendliness in programming parallel graph algorithms, we first summarize the programming abstractions (languages and models) of existing systems. While most systems adopt existing programming languages that are familiar to users (e.g., C/C++ and Java), some systems require users to learn a new domain-specific language dedicated to programming parallel graph algorithms (e.g., Green-Marl [Hong et al., 2012] and Trinity Specification Language [Shao et al., 2013]).

1.2.1 Programming Model

Most big graph systems adopt the vertex-centric model where a programmer only needs to specify the behavior of one vertex. The vertex-centric model can be further divided into two types: (1) message passing (e.g., in Pregel), where vertices communicate with each other by sending messages; and (2) shared-memory abstraction (e.g., in GraphLab), where vertices directly access the states of other vertices and edges.

Message passing is a natural model in a distributed environment, since users can explicitly dictate message passing behavior in their programs. In contrast, the shared-memory abstraction allows programmers to directly access data as if operating on a single machine, and most single-machine vertex-centric systems adopt this model. However, distributed GraphLab adopts the shared-memory abstraction and there are also single-machine systems that adopt message passing (e.g., Flash-Graph [Zheng et al., 2015]).

The vertex-centric framework can be further extended with a block-centric model (e.g., Giraph++ [Tian et al., 2013] and Blogel [Yan et al., 2014a]), which partitions the vertices into multiple disjoint subgraphs, so that value propagation within each subgraph could bypass network communication. The block-centric model often improves the performance of graph computation by orders of magnitude.

Besides the vertex-centric systems, some big graph systems adopt a matrix-based programming model; these include PEGASUS [Kang et al., 2009], GBASE [Kang et al., 2011], and SystemML [Ghoting et al., 2011]. These systems represent a graph algorithm by a sequence of generalized matrix-vector multiplications, which can be efficiently processed since sparse matrix algebra has been studied for decades in the High Performance Computing (HPC) field. However, users who are not familiar with matrix algebra might prefer vertex-centric programming to matrix-based programming. Recently, Sundaram et al. [2015] helped bridge the gap for these users: their GraphMat system translates a vertex-centric program to high performance sparse matrix operations to be run on the backend.

Another important class of programming models is subgraph-centric models, where users write programs to process a subgraph in-

stead of a single vertex. These models target graph problems whose output size can be exponential to the graph size (e.g., graph matching and finding motifs), or problems that require analyzing entire neighborhoods in a holistic manner. Since vertices in a subgraph can be randomly accessed by a user program, a critical issue for a subgraph-centric model is how to efficiently construct the relevant subgraphs. Arabesque [Teixeira et al., 2015] and NScale [Quamar et al., 2016] are two systems that use a subgraph-centric model, although there are significant differences in the models they adopt.

There are also systems that require users to write graph algorithms using a domain specific language (DSL), e.g., Green-Marl [Hong et al., 2012, 2014], Galois [Nguyen et al., 2013], and Ligma [Shun and Blelloch, 2013]. The language constructs of those DSLs expose opportunities for parallelism, which can be utilized by the system for efficient parallel execution. Of course, users have to learn a new language or programming paradigm in order to use such a system.

Finally, several recent systems have been built to bring in declarative query languages for big graph analytics. First, since many graph algorithms can be expressed as recursive Datalog [Bancilhon and Ramakrishnan, 1986] queries, a number of research projects are inventing new-generation Datalog systems for scalable big graph analytics. Second, often times, a graph analytics job is only one part of a gigantic, end-to-end SQL³-dominated data analysis pipeline which includes constructing graphs dynamically from tabular data sources and converting graph computation results back into tabular reports; therefore, several systems have integrated vertex-centric programming models into declarative query languages to make those end-to-end data analysis tasks easier [Simmen et al., 2014, Gonzalez et al., 2014].

1.2.2 Expressiveness

Most big graph systems aim at solving a broad class of graph problems using a unified programming framework. Therefore, it is meaningless to study big graph systems without studying the algorithms and applications that can be implemented in these systems. However, many

³SQL. <https://en.wikipedia.org/wiki/SQL>

papers just introduce API simplicity and performance advantages of their systems in order to promote their work, but these benefits may come at a cost of additional assumptions and narrower expressiveness that were understated, which should be made clear to avoid blind or even wrong system choice. We now discuss the expressiveness of the various programming models described before, and provide some advice on how to choose an appropriate framework for an application at hand.

Many graph algorithms only require each vertex to communicate with its neighbors, such as PageRank and other more complicated random walk algorithms (e.g., [Zhang et al., 2016]). In these algorithms, intermediate data are only exchanged along edges, and so the volume of intermediate data is comparable to the data size. We say that these algorithms require *edge-based communication*. In some of these algorithms, a vertex only needs the aggregated value of the received values, which provides opportunities for further optimization. For example, MOCgraph [Zhou et al., 2014], GraphD [Yan et al., 2016d], and the superstep-splitting technique of Giraph [Ching et al., 2015] all propose aggregating messages earlier instead of buffering them for later processing, in order to save memory space; while PowerGraph [Gonzalez et al., 2012], GraphChi [Kyrola et al., 2012] and X-Stream [Roy et al., 2013] assume that data values are aggregated at each vertex from its incoming edges, in their model design. We, however, would like to indicate that not all algorithms with edge-based communication allow its vertices to aggregate received values, such as the attribute broadcast algorithm of Yan et al. [2015].

Edge-based communication implies that any information can be propagated for just one hop at a time, which leads to poor performance if a vertex u needs to transmit a value to another vertex v far away from u in a large-diameter graph. Pointer jumping (aka path doubling), a technique from PRAM algorithm design, solves this problem by doubling the propagation length from u to v , until v is reached. This requires a vertex to be able to send data to any other vertex, not just its neighbors. We say that these algorithms require *ID-based communication*, where a vertex u can send messages to another vertex w as long

as w 's ID is known. Pregel [Malewicz et al., 2010] adopts ID-based communication and thus can implement pointer-jumping algorithms such as those to be described in Chapter 3.2, while GraphLab [Gonzalez et al., 2012] only allows each vertex to access its neighbors' data, and thus cannot support these algorithms. In fact, Pregel has probably the most expressive programming model in theory, and it is known how to write a large number of graph algorithms efficiently in that model. The Bulk Synchronous Parallel (BSP) model, on which Pregel is based, has been very well-studied, but as a synchronous model, the number of iterations must be kept low in a distributed setting, which can be achieved with the help of pointer jumping.

Another solution to avoid slow value propagation is to use a block-centric model, where nearby vertices are grouped into a block for processing together each time. In a distributed environment, since a block is assigned to a unique machine, only blocks need to communicate with each other, and computation over vertices inside a block does not generate communication. In a single-machine environment, each block usually fits in a CPU cache, and thus block-based processing improves cache locality in its execution. In addition to faster value propagation (i.e., block-wise), the block-centric model also significantly reduces the communication workload. Representative block-centric systems include Giraph++ [Tian et al., 2013] and Blogel [Yan et al., 2014a].

Some graph algorithms (e.g., k -core finding [Salihoglu and Widom, 2014]) need to mutate the graph topology during computation, and thus, support for deletion and addition of edges and vertices is also an important aspect of system expressiveness. For example, VENUS [Cheng et al., 2015] streams immutable graph structure and thus does not support algorithms that require graph mutations.

The models discussed so far are mostly vertex-centric. However, many graph mining problems define constraints on subgraphs, e.g., graph matching and motif mining. Subgraph-based models are proposed to solve these problems by writing user-friendly programs, where computation is directly performed on subgraphs. Such systems include NScale [Quamar et al., 2016] and Arabesque [Teixeira et al., 2015], which we discuss in more detail in Chapter 7.

We remark that there are other models that could be more appropriate for a specific application at hand. For example, if one is viewing a graph as a matrix, and solving a machine learning problem that uses matrix operations, then matrix-based systems like SystemML [Ghoting et al., 2011] could be a better choice. Also, if graph processing is just part of a dataflow program, then dataflow-based systems could provide more flexibility, e.g., GraphX [Gonzalez et al., 2014] can interoperate with other dataflow operators in Spark [Zaharia et al., 2012] to avoid data import/export.

1.2.3 Execution Mode

Most big graph systems target iterative graph computation, where vertex values are repeatedly updated until the computation converges. There are two typical execution modes: synchronous and asynchronous. The synchronous mode is also called bulk synchronous parallel (BSP), exemplified by Pregel, while the asynchronous mode is adopted by GraphLab and several other systems (especially those targeting machine learning workloads). The difference between these two modes is that, in the synchronous mode, there is a global barrier from one iteration to another, and out-going messages or updates of one iteration are only accessible in the next iteration; in the asynchronous mode, a vertex has immediate access to its in-bound messages or updates.

Asynchronous parallel computation incurs race conditions and thus requires additional effort to enforce data consistency (e.g., by using locks). Moreover, in a distributed environment, asynchronous execution tends to transmit a lot of small messages, since the update to a vertex value should be reflected in time. In contrast, BSP only requires updates to be synchronized at the end of each iteration, and messages can be sent in large batches. In fact, GraphLab has a synchronous mode that simulates the BSP mode of Pregel, and both Lu et al. [2014] and Han et al. [2014a] found that the synchronous mode is generally faster than the asynchronous mode. Further, for many algorithms, asynchronous execution is not an option because the indeterministic execution may lead to incorrect answers.

However, for some problems like PageRank computation, vertex values converge asymmetrically: most vertices converge quickly after a few iterations, but some vertices take a large number of iterations to converge. In that case, asynchronous execution can schedule those vertices that converge more slowly to compute for more iterations, while synchronous execution processes every vertex once in each iteration even if most vertices are converged. Therefore, asynchronous mode is much faster for such algorithms and is thus preferred. Moreover, asynchronous computation is always preferred in a single-machine system since data access no longer incurs network communication, and accessing the latest vertex value leads to faster convergence.

It is, however, worth noting that some asynchronous frameworks may not converge to the exact results (e.g., PageRank values), but the approximate results are often good enough while the significant improvement in performance (compared with synchronous execution) is highly attractive. More discussion can be found in Section 4.2.

Recently, PowerSwitch [Xie et al., 2015a] showed how to support mode switch between asynchronous execution and synchronous execution in GraphLab. They found that when the workload is low, asynchronous execution is faster due to the faster convergence rate provided by accessing the latest values. Race conditions (e.g., updates to the same vertex) are unlikely to occur since only a small portion of vertices participate in computation, and the number of messages is too small to benefit from sending in large batches. In contrast, when the workload is high, synchronous execution is faster since there is no need to handle race conditions (i.e., it avoids the expensive locking/unlocking cost required by asynchronous execution), and messages are sent in large batches. Thus, PowerSwitch constantly collects execution statistics on-the-fly, which are used to predict future performance and determine the timing of a profitable mode switch.

1.2.4 Other Features

There are also many other dimensions to categorize big graph systems. As for the **execution environment**, there are systems developed to process graphs in a single machine, or using a cluster of machines.

The single-machine environment can be further divided into two types, commodity PCs and high-end servers. The former targets processing big graphs efficiently using readily available resources; since the available memory on a commodity PC is limited, the graph is usually disk-resident, and loaded into memory for processing part-by-part or in a streaming fashion. The latter aims at beating distributed systems by eliminating the cost of network communication, and the graph is usually memory-resident. As for the **graph placement**, distributed systems usually keep the graph in main memory, since there are many machines and the total RAM size is sufficient, while single-PC systems tend to process disk-resident or SSD-resident graphs. There are also distributed systems that process disk-resident graphs in order to scale to giant graphs whose size is much larger than the total RAM size in a cluster, such as Pregelix [Bu et al., 2014], GraphD [Yan et al., 2016d] and Chaos [Roy et al., 2015].

There are also many design techniques that may significantly influence the system performance for specific algorithms. For example, disk-based single-machine systems like GraphChi [Kyrola et al., 2012] are designed for iterative batch processing, while TurboGraph [Han et al., 2013] maintains an in-memory page ID table for directly locating the disk page of any vertex. Given these differences in system design, a reader will not be surprised to see a claim like TurboGraph “significantly outperforms GraphChi by up to four orders of magnitude”, for a query that is to find the neighbors of a particular vertex.

1.3 Organization of the Survey

The diverse features supported by different big graph systems, and the cross-cutting nature of many of the key designs, make it challenging to organize such a survey. As an example, the popular vertex-centric programming model is easy to support on top of a wide range of different underlying implementations, including distributed frameworks like Hadoop MapReduce, matrix-based systems, and relational databases. However, each of those implementations raises unique and different challenges despite their use of the vertex-centric model on top.

In this survey, we focus on presenting the key designs and features of the various graph processing systems, while endeavoring to place related systems together to summarize the common ideas underlying their designs. For quick reference, Table 1.1 presents a list of all the systems that we discuss in each chapter.

We broadly divide the survey into three parts. In Part I, we discuss the big graph systems that primarily use the vertex-centric programming model, which has been widely studied recently due to its simplicity in programming parallel graph algorithms. Specifically, Chapter 3 reviews the framework of Pregel, and introduces how to develop algorithms with performance guarantees in Pregel; it then discusses existing open-source Pregel-like systems with improvements in communication mechanism, load balancing, out-of-core support and fault tolerance. Chapter 4 walks through the various extensions to the basic framework of Pregel that could significantly improve the performance of graph computations. Chapter 5 covers a few important big graph systems that adopt shared memory abstraction, including the pioneering GraphLab system.

In Part II, we review other systems that attempt to provide support for more general graph programming models; most of these are motivated by the observation that complex graph algorithms or analysis tasks are often difficult to program using the simple vertex-centric programming framework. Chapter 6 describes several matrix-based big graph systems, including the pioneering MapReduce-based systems PEGASUS and GBASE, and the more powerful SystemML system. Chapter 7 explains why the vertex-centric and matrix-based frameworks are not sufficient for graph problems like graph matching and motif mining, and introduces two subgraph-centric systems, NScale and Arabesque, to process such graph problems efficiently. Chapter 8 reviews several systems that either offer database-style declarative query languages or leverage database-style query processing techniques.

Finally, in Part III, we discuss some miscellaneous issues. While some vertex-centric single-machine big graph systems are also introduced in Chapter 5, Chapter 9 surveys more single-machine systems that adopt a computation model beyond a pure vertex-centric one.

Table 1.1

Section	System
3.1	Pregel
3.3	Giraph, Pregel+, GPS, MOCgraph
3.4	WindCatch, PAGE
3.5	GraphD
4.1	Giraph++, Blogel
4.2	Maiter, GiraphUC
4.3	Quegel
5.1	GraphLab/PowerGraph
5.2	GraphChi, X-Stream, Chaos, VENUS, GridGraph
6.1	PEGASUS
6.2	GBASE
6.2	SystemML
7.1.1	Trinity
7.2	NScale
7.3	Arabesque
8.1	Socialite, DeALS, Myria, Yedalog
8.2	GraphX, Pregelix, Vertexica
8.3	REX, Maiter
8.4	Aster Data
9.1	GraphMat, GraphTwist
9.2	Green-Marl, Ligra, GRACE, Galois
10.1	TurboGraph, FlashGraph
10.2	Medusa, MapGraph, CuSha
11.2	Chronos, DeltaGraph, LLAMA
11.3	Kineograph, TIDE

Chapter 10 discusses a few systems that utilize new hardware technologies to significantly boost the performance of big graph analytics. Then, in Chapter 11, we discuss the issues of managing time-evolving graphs and supporting real-time analytics over streaming graph data, and discuss several recent systems that focus on providing those capabilities. Finally, we conclude the survey in Chapter 12 and provide a discussion on future research in big graph analytics platforms.

2

Preliminaries

In this chapter, we present the architectures of typical single-machine and distributed graph analytics systems, and briefly discuss some of the key considerations; we also discuss some of the techniques used to improve the performance and robustness of such systems.

2.1 Data Models and Analytics Tasks

We begin with introducing some of the notation used throughout this survey. In this survey, we consider an input graph $G = (V, E)$ where each vertex $v \in V$ has a unique ID $id(v)$. For simplicity, we use v and $id(v)$ interchangeably. The number of vertices and edges are denoted by $|V|$ and $|E|$, respectively. If G is undirected, we denote the set of neighbors of v by $\Gamma(v)$, and denote the degree of v by $d(v)$. If G is directed, we denote the set of in-neighbors (and out-neighbors) of v by $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, and denote the in-degree (and out-degree) of v by $d_{in}v$ (and $d_{out}(v)$). The graph diameter is denoted by δ . Both vertices and edges may have attributes associated with them, such as edge weights or text labels.

A graph can be equivalently seen as a $|V| \times |V|$ *adjacency matrix*, where each non-zero matrix entry implies the existence of an edge between the corresponding nodes in the graph. This equivalence allows using matrix operations and libraries to execute graph analytics tasks [Kepner and Gilbert, 2011].

This survey reviews general-purpose systems for running a graph analytics task that takes a graph as input, and produces an output which can be: (a) one or a small number of scalar values (e.g., global clustering coefficient of the graph), (b) a set of values, one per node/edge (e.g., PageRank of each node in the graph), (c) a set of values, one for each instance of a subgraph in the graph, with subgraphs of interest specified in some fashion (e.g., finding maximal cliques), or (d) other forms such as in graph summarization and deduplication. Graph analysis tasks of Category (b) are perhaps the most prevalent, and certainly the most well-studied from systems perspective. Note that, different analysis tasks from this class may exhibit very different computational complexity; e.g., PageRank and *betweenness centrality* are both centrality measures whose computation falls in this class, but while PageRank computation is relatively easy to parallelize and execute, betweenness centrality is much harder to compute.

We can make a further distinction between one-shot analytics and continuous analytics. In the latter case, the input graph itself is dynamically changing over time, and the goal is to execute the analytics task continuously or periodically. Most of the work in big graph analytics systems has focused on one-shot analytics; we discuss the work on continuous analytics towards the end of the survey.

Finally, an important consideration is how the graph is physically represented, and where it is stored. Most of the systems that we cover assume that the graph is stored on secondary storages (either a distributed file system like HDFS, or a local file system), in either human-readable text format or serialized binary format. Some systems expect the graph to be provided as a list of edges, whereas others may require an adjacency list representation, with all the edges of a vertex listed along with the vertex information. Many of the systems can also ingest data from other data stores like key-value stores, or relational

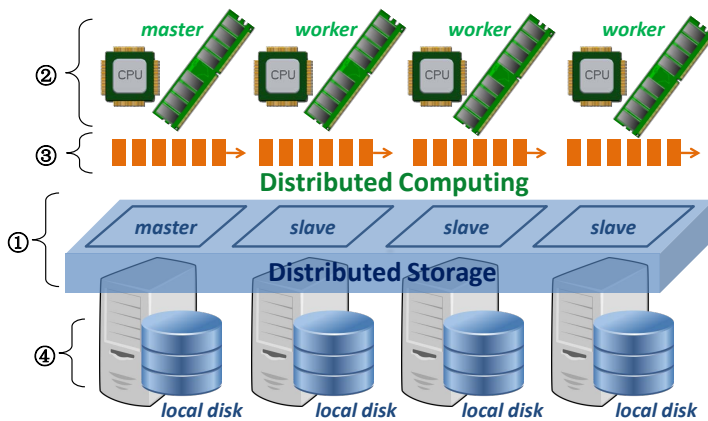


Figure 2.1: Components of a Distributed System

databases. In many cases, the graph may itself have to be generated by processing other non-graph datasets [Xirogiannopoulos et al., 2015]; although this is an important practical consideration, there is little systematic work on it so far and we do not discuss it further in this survey.

2.2 Distributed Architecture

Next we discuss the components of a typical distributed graph analytics system, and some of the key decisions that significantly impact the performance of such a system.

Components. The typical architecture of a distributed graph analytics system is shown in Figure 2.1, which consists of four components.

Component ① is an underlying distributed file storage for keeping graph datasets and analytics results, which is usually resilient to machine failures. Most graph analytics systems such as Giraph [Ching et al., 2015] and GraphLab [Gonzalez et al., 2012] support loading data from HDFS, which by default replicates each data block on three machines so that the failure of any two machines will not cause data loss. Besides HDFS, the storage component can also be other distributed

key-value stores or databases, such as HBase¹ and Cassandra². There also exist distributed systems designed with their own storage sub-systems, e.g., Chaos [Roy et al., 2015], but this loses the interoperability among various Big Data frameworks, and also incurs expensive preprocessing cost of data conversion.

Component ② is the computation module that loads a graph dataset from the underlying distributed storage, and performs the actual computation. The key design issue of this component is how to expose a user-friendly programming interface to users, hiding low-level details of parallel computation. Various computation models have been designed as we survey in depth later.

Component ③ is the communication layer that is used by the computation module for communication between machines. The goal of this component is to expose a simple and clean communication interface to the computation module, in order to simplify the design of the computation module. This component does not directly interact with application developers (users), since a user-friendly programming interface should hide details like network communication from the users. Different systems use different communication mechanisms. For example, Giraph [Ching et al., 2015], Pregel+ [Yan et al., 2015], and GraphLab [Gonzalez et al., 2012] use Netty³, MPI and Remote Procedure Call (RPC), respectively.

Component ④ is local disk storage, which is managed by the respective machines individually, to offload some data from memory to local disk(s) during the computation. This is usually used by out-of-core systems (or modes) such as GraphD [Yan et al., 2016d], Pregelix [Bu et al., 2014], and out-of-core Giraph. Some systems even offload data to HDFS and only exchange control information among the machines, an example being epiC [Jiang et al., 2014], but this may not be a good solution since data offloading incurs network communication which is often more expensive than local disk IO (e.g., HDFS always replicates data to other machines for fault tolerance).

¹<http://hbase.apache.org/>

²<http://cassandra.apache.org/>

³<http://netty.io/>

Graph Partitioning. To execute the graph analysis task in a parallel fashion, the graph must be *partitioned* (*sharded*) across the machines in the cluster. The graph analytics system typically creates a set of partitions using some policy, which are then transparently mapped to the actual machines by the underlying framework (e.g., Yarn or Mesos); some systems explicitly “over-partition” to simplify load-balancing, in which case multiple logical partitions are mapped to the same physical machine. However, we often talk about mapping of a vertex to a machine to simplify discussion.

Most systems partition the graph by vertices, i.e., each partition consists of a set of vertices, all the properties of those vertices, and the outgoing edges for those vertices. A straightforward method is to partition the vertices by their IDs. For example, hash-based partitioning computes the partition for a vertex using a hash function that takes the vertex ID as input, while range-based partitioning divides vertices into different partitions by ID ranges. The benefit of this method is that we can compute the partition of a vertex (for sending messages to that partition) in $O(1)$ time. Optionally, one may create partitions with the goal of reducing the number of edges that are cut (and thus the total communication during analysis); however, this comes with the additional cost of having to run an expensive graph partitioning algorithm. This approach also makes it more costly to find the location of a vertex, and typically a distributed hash table needs to be used for this purpose incurring additional communication (and thus round-trip delay) [Shang and Yu, 2013, Khayyat et al., 2013]; one may also recode IDs in a pre-processing step like in Giraph++ [Tian et al., 2013]. Shao et al. [2015] even observed that Giraph exhibits worse performance when a better partitioning is provided, as there are insufficient processors to process local messages. Maintaining such partitions in presence of dynamic updates to the graph is quite tricky, and typically this option is not used in such cases.

Instead of partitioning by vertices, several systems advocate partitioning by edges instead [Gonzalez et al., 2012]. In such cases, the edges of a specific vertex v may be distributed to multiple machines, each also keeping a replica of v ’s data. This approach often leads to a more bal-

anced workload across the machines, especially given that many graphs (e.g., social networks) obey power-law degree distribution, resulting in a few very high-degree vertices.

Synchronous vs Asynchronous Execution. Another issue is whether the execution is synchronous or asynchronous. Consider PageRank computation, where each vertex keeps collecting values from in-neighbors and distributing values to out-neighbors until its PageRank converges. Since different vertices have different convergence rates, asynchronous execution allows those vertices that converge slowly (resp., quickly) to run for more (resp., fewer) rounds, and is thus more efficient. On the other hand, synchronous execution has more deterministic behavior, and it is easier to *batch* messages there to reduce the overall communication cost. Usually asynchronous execution is managed by a (distributed) prioritized scheduler that allows prioritized execution, while guaranteeing sufficient parallelism. One method to determine priority is delta propagation, where vertices propagate values (or deltas) to neighbors according to the incremental values (or deltas) it receives, and a vertex with a large delta has a higher priority of execution [Zhang et al., 2014]. In contrast, to perform breadth-first search (BFS), synchronous execution where the i -th round activates the vertices i hops away from the source vertex guarantees minimum workload. Synchronous execution is also easy to analyze and debug due to its deterministic behavior, and avoids race conditions and thus the locking/unlocking cost. To sum up, synchronous execution is a desirable choice unless the algorithm has an asymmetric convergence behavior.

Fault Tolerance. The last issue we would like to mention is fault tolerance. Although the underlying distributed storage (e.g., HDFS) already provides resilience to data loss, if a long-running job fails, reloading data and computing from scratch wastes all the previous computation. Checkpointing provides a trade-off between (1) the failure-free cost paid for being fault-tolerant, and (2) fast recovery. During synchronous execution, all machines can periodically suspend their computation at synchronization barriers, to write the current state of computation to the resilient distributed storage. If machine failure happens later, the latest stored computation state can be loaded to continue execu-

tion (rather than starting from scratch). This method is called coordinated checkpointing. There also exist uncoordinated checkpointing methods [Chandy and Lamport, 1985] that do not need to suspend execution, which is ideal for asynchronous execution. For algorithms that always converge to a fixed point, checkpointing can be avoided and recovery can be done by re-initializing the states of the lost data [Schelter et al., 2013].

2.3 Single-Machine Architecture

A single-machine graph analytics system is designed for running with a single machine, thus avoiding the expensive network communication overhead; however, such a system cannot scale out and thus has fixed hardware resources. As a result, the running time is usually proportional to the graph size. There are two types of single-machine systems: (1) for commodity PCs with limited resources (especially memory), and (2) for servers with many cores and a large memory.

For simplicity, let us assume that the computation is performed at the granularity of vertices. A typical single-machine system will partition vertices into multiple vertex shards $V_1, V_2, \dots, V_n$, and each vertex shard V_i is associated with an edge shard E_i , which keeps the edges (e.g., adjacency lists) of vertices in V_i . For Type (1) systems, partitioning is performed so that each shard can be loaded into memory for computation at a time. Some systems like GraphChi [Kyrola et al., 2012] load both V_i and E_i into memory for processing, while others such as X-Stream [Roy et al., 2013] and VENUS [Cheng et al., 2015] load only V_i into memory and stream E_i from disk. For Type (2) systems, partitioning provides sufficient parallelism so that each shard is processed by one core. Some of the latter systems may not even create explicit shards, but rather load the entire graph into the shared memory, and parallelize in an adaptive manner based on the specific computation being performed.

Compared with a distributed architecture, a single-machine system may need to preprocess the graph by sharding, which may not pay off if only one light-workload job needs to be run subsequently.

In fact, most such systems simply use a straightforward range-based vertex partitioning [Kyrola et al., 2012, Roy et al., 2013, Cheng et al., 2015], rather than more sophisticated ones for sharding efficiency. A Type (2) single-machine system also requires one machine to load the entire graph, which can be much more costly than the parallel loading in a distributed system.

In general, we observe that a single-machine system can be more efficient for small graphs and computationally-light jobs, since there is no communication overhead. After all, distributed execution incurs round-trip delay, the cost of which could be already more than the actual computation. In contrast, a distributed system is more appropriate for big graphs and computationally-heavy jobs, since the scale-out solution requires each machine to process only a portion of the entire graph.

To better reason about efficiency of single-machine or distributed systems, McSherry et al. [2015] proposed a metric called COST (Configuration that Outperforms a Single Thread). For a given system and for a given problem, the COST is the hardware configuration (measured in the number of cores) required before the system outperforms a competent single-threaded implementation (which may be able to use a better, but non-parallelizable algorithm). Thus, the metric weighs the scalability of a system against the overheads introduced by the system to achieve that scalability. Their studies of popular systems like Giraph, GraphX and GraphLab show that those systems have very high COSTs, and often seem to perform worse than a single-threaded implementation despite using many more cores.

We caution that COST should only be used as one of the criteria to compare the pros and cons of different systems, and as a sanity check on the overheads. A single-threaded implementation is fundamentally limited in its ability to exploit parallelism. Further, for many graph analysis tasks, an easy-to-use programming framework that interacts with other widely used frameworks (like key-value stores, HDFS, etc.), is perhaps a more important criteria than pure performance.

Part I

Vertex-Centric Programming Model



3

Vertex-Centric Message Passing (Pregel-like) Systems

In this chapter, we first review the framework of Pregel in Section 3.1, and introduce how to develop Pregel algorithms with performance guarantees in Section 3.2. We then introduce the existing Pregel-like systems with improvements in communication mechanism (Section 3.3), load balancing (Section 3.4), out-of-core support (Section 3.5) and fault tolerance (Section 3.6).

3.1 The Framework of Pregel

Computation and Programming Model. A Pregel job starts by loading the input graph G from GFS (Google File System) into the main memories of the worker machines (or workers) in a cluster, where vertices are partitioned among the workers. Typically, vertices are partitioned by a hash function $hash(.)$ known by all workers: each vertex v is assigned to a worker $W = hash(v)$. Each vertex v maintains its adjacency list (which usually stores $\Gamma(v)$ or $\Gamma_{out}(v)$), a vertex value $a(v)$, and also a flag $active(v)$ indicating whether v is active or halted.

A Pregel job proceeds in iterations, where an iteration is also called a superstep. In Pregel, a user needs to specify a user-defined func-

tion (UDF) $compute(msgs)$ to be called by a vertex v , where $msgs$ is the set of incoming messages sent to v in the previous superstep. In $v.compute(.)$, v may update $a(v)$, send messages to other vertices, and vote to halt (i.e., deactivate itself). Only active vertices will call $compute(.)$ in a superstep, but a halted vertex will be reactivated if it receives a message. The program terminates when all vertices are halted and there is no pending message for the next superstep. Finally, the results are dumped to GFS.

We now illustrate how to write $compute(.)$, using two graph algorithms. In these algorithms, a vertex only sends messages to its neighbors (or out-neighbors), whose IDs are directly available in its adjacency list. In Section 3.2, we will see some Pregel algorithms where a vertex sends messages to non-neighbors.

Example 1: PageRank. Consider the PageRank algorithm of [Malewicz et al., 2010] where $a(v)$ stores the PageRank value of vertex v , and $a(v)$ gets updated until convergence. In Step 1, each vertex v initializes $a(v) = 1/|V|$ and distributes $a(v)$ to its out-neighbors by sending each out-neighbor a message $a(v)/d_{out}(v)$. In Step i ($i > 1$), each vertex v sums up the received message values, denoted by sum , and computes $a(v) = 0.15/|V| + 0.85 \cdot sum$. It then distributes $a(v)/d_{out}(v)$ to each of its out-neighbors. If we want to perform PageRank computation for n supersteps, we can let every vertex vote to halt and exit $compute(.)$ in Step $(n + 1)$.

Example 2: Hash-Min. We consider the Hash-Min algorithm of [Yan et al., 2014b] for computing the connected components (CCs) of an undirected graph G . Given a CC C , we denote the set of vertices of C by $V(C)$, and define the ID of a CC C to be $cc(v) = \min\{id(u) : u \in V(C)\}$. Hash-Min computes $cc(v)$ for each vertex $v \in V$. The idea is to broadcast the smallest vertex ID seen so far by each vertex v , which is stored in $a(v)$. In Step 1, each vertex v sets $a(v)$ to be the smallest ID among $id(v)$ and $id(u)$ of all $u \in \Gamma(v)$, broadcasts $a(v)$ to all its neighbors, and votes to halt. In Step i ($i > 1$), each vertex v receives messages (if any) from its neighbors; let min be the smallest ID received, if $min < a(v)$, v sets $a(v) = min$ and broadcasts $a(v)$ to

its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, $a(v) = cc(v)$ for all v .

We call the case where most vertices participate in the computation as **dense vertex access**, and the case where only a small fraction of vertices participate in the computation as **sparse vertex access**. Every superstep of the PageRank algorithm is dense, while for Hash-Min, earlier supersteps are dense while later supersteps are sparse.

Combiner. To reduce the number of messages transferred though the network, users may implement a message combiner to specify how to combine messages targeted at the same vertex v_t , so that messages on a worker W targeted at v_t will be combined into a single message by W locally, and then sent to v_t . In the PageRank (resp. Hash-Min) algorithm, the combiner can be implemented as the summation (resp. minimum) operation, since only the summation (resp. minimum) of incoming messages is of interest in *compute(.)*.

Aggregator. Pregel also allows users to implement an aggregator for global communication. Each vertex can provide a value to an aggregator in *compute(.)* in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep. In the actual implementation, the values are first aggregated locally on each worker; then, the aggregated values are then aggregated globally at the master; the globally aggregated value is then broadcast back to all workers.

Fault Tolerance. To be fault tolerant, a Pregel job may be specified to periodically back up the state of computation to GFS at superstep boundaries as a checkpoint (e.g., every 10 supersteps). Since data on GFS is replicated on multiple machines, a checkpoint is resilient to machine failures. If a failure happens, all workers simply reload the state of computation from the latest checkpoint and then continue computation from the last checkpointed superstep.

Graph Mutations. Pregel also supports graph mutations, which are categorized into two types: (1) local mutations, where a vertex adds or removes its own edges or removes itself; and (2) global mutations, where

a vertex adds or removes (i) the edges of other vertices or (ii) other vertices. Global mutations may incur conflicts and users may specify conflict resolution policies to avoid nondeterministic behavior. In a superstep, edge removals are performed first, and then vertex removals, followed by vertex addition, and finally edge addition.

Graph Mutations are useful in some graph algorithms, such as the algorithm for k -core finding [Quick et al., 2012] which finds the maximal subgraphs of an undirected graph G in which every vertex has a degree of at least k . In each superstep, the algorithm lets each vertex whose degree is less than k delete itself and its adjacent edges, until all the remaining vertices have degree at least k .

3.2 Algorithm Design in Pregel

Although there is an abundance of papers studying system improvements to Pregel, the number of papers studying algorithm design in Pregel is still very limited. However, as a general-purpose graph processing framework, it is important to study how to design scalable algorithms on top of Pregel.

There are a few papers studying Pregel algorithms: [Quick et al., 2012] demonstrated that many social network analytic tasks can be formulated as Pregel algorithms, while [Salihoglu and Widom, 2014] proposed four algorithm-specific techniques to improve the performance of some Pregel algorithms. However, these algorithms are still ad-hoc and there lacks any cost analysis.

We remark that it is important for users to be aware of the scalability of a Pregel algorithm. For example, in the triangle finding algorithm of [Salihoglu and Widom, 2014], assuming that $v_2, v_3 \in \Gamma(v_1)$; then, to determine whether a triangle $\triangle v_1 v_2 v_3$ exists, vertex v_1 sends a message to v_2 inquiring whether $v_3 \in \Gamma(v_2)$. Since there are $O(|E|^{\frac{3}{2}})$ triangles in a graph G , the number of messages generated in a superstep can be much larger than the graph size, leading to long-running supersteps or even memory overflow (note that Pregel buffers messages in main memories to be processed by `compute(.)`). If a user is aware of this scalability issue, he/she may simply design the algorithm to send inquiries

for only a small subset of vertices in each superstep, so as to avoid memory overflow.

Practical Pregel Algorithm (PPA). [Yan et al., 2014b] identified a class of Pregel algorithms that have good scalability, called practical Pregel algorithms (PPAs). Specifically, a Pregel algorithm is called a **balanced practical Pregel algorithm (BPPA)** if it satisfies the following constraints:

1. *Linear space usage:* each vertex v uses $O(d(v))$, or $O(d_{in}(v) + d_{out}(v))$, memory space.
2. *Linear computation cost:* the time complexity of $v.compute(.)$ is $O(d(v))$, or $O(d_{in}(v) + d_{out}(v))$.
3. *Linear communication cost:* at each superstep, the volume of the messages sent/received by each vertex v is $O(d(v))$, or $O(d_{in}(v) + d_{out}(v))$.
4. *At most logarithmic number of rounds:* the algorithm terminates after $O(\log |V|)$ supersteps.

Constraints 1-3 offers good load balancing and linear cost at each superstep, while Constraint 4 controls the total running time (note that each superstep requires a global barrier at the end, which incurs some cost). For some graph problems, the vertex-grained requirements of BPPA can be too strict, and we can only achieve *overall linear space usage, computation cost, and communication cost* (still in $O(\log |V|)$ rounds). For example, in the S-V algorithm to be discussed at the end of this section, some vertices may send many more messages than their degrees, but the total number of messages is still linear to the graph size. We call a Pregel algorithm that satisfies these constraints simply as a **practical Pregel algorithm (PPA)**.

The Hash-Min algorithm presented in Section 3.1 is not a PPA, since it takes $O(\delta)$ supersteps. For example, in a path graph where the smallest vertex is on one end of the path, Hash-Min runs for $|E|$ supersteps, the cost of which is prohibitive since each superstep requires message synchronization. To achieve the $O(\log |V|)$ -superstep

bound, it is usually necessary to use the pointer jumping (a.k.a. path doubling) technique where a vertex needs to send messages to non-neighbors. Pointer jumping is used in many Pregel algorithms, such as the S-V algorithm for computing connected components (CCs) [Yan et al., 2014b], the Pregel algorithm for computing biconnected components [Yan et al., 2014b], and the Pregel algorithm for computing minimum spanning forest [Salihoglu and Widom, 2014]. We illustrate the idea of pointer jumping using two PPAs, one for list ranking and the other for computing CCs.

Example 3: List Ranking. The list ranking problem is as follows, where we assume elements in a list are linked backwards from the tail to the head (the algorithm for the forward linking order can be similarly derived). Consider a linked list $\mathcal{L}$ with n vertices, where each vertex v is associated with a value $val(v)$ and a link to its predecessor $pred(v)$ (i.e., $a(v)$ consists of $val(v)$ and $pred(v)$). The vertex v at the head of $\mathcal{L}$ has $pred(v) = null$. For each $v \in \mathcal{L}$, let us define $sum(v)$ to be the sum of the values of all the vertices from v following the predecessor link to the head. The goal is to compute $sum(v)$ for every $v \in \mathcal{L}$. If $val(v) = 1$ for any $v \in \mathcal{L}$, then $sum(v)$ is simply the rank of v in the list, i.e., the number of vertices preceding v plus 1. Note that in the input data, vertices are in arbitrary order. Albeit simple, list ranking is important in parallel graph computation; for example, it is a building block of the PPA for computing biconnected components [Yan et al., 2014b].

We now describe a BPPA for list ranking. Initially, each vertex v assigns $sum(v) \leftarrow val(v)$. Then in each round, each vertex v does the following: If $pred(v) \neq null$, v sets $sum(v) \leftarrow sum(v) + sum(pred(v))$ and $pred(v) \leftarrow pred(pred(v))$; otherwise, v votes to halt. The if-branch is accomplished in three supersteps: (1) v sends a message (whose value is its own ID) to $u = pred(v)$ requesting for the values of $sum(u)$ and $pred(u)$; (2) u sends back the requested values to each requesting vertex v ; and (3) v updates $sum(v)$ and $pred(v)$ using the received values. This process repeats until $pred(v) = null$ for every vertex v , at which point all vertices vote to halt and we have $sum(v)$ as desired.

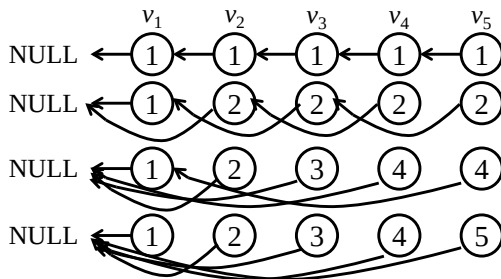


Figure 3.1: Illustration of the BPPA for List Ranking

Figure 3.1 illustrates how the algorithm works. Initially, objects v_1 – v_5 form a linked list with $sum(v_i) = val(v_i) = 1$ and $pred(v_i) = v_{i-1}$. Let us now focus on v_5 . In Round 1, we have $pred(v_5) = v_4$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_4) = 1 + 1 = 2$ and $pred(v_5) \leftarrow pred(v_4) = v_3$. One can verify the states of the other vertices similarly. In Round 2, we have $pred(v_5) = v_3$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_3) = 2 + 2 = 4$ and $pred(v_5) \leftarrow pred(v_3) = v_1$. In Round 3, we have $pred(v_5) = v_1$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_1) = 4 + 1 = 5$ and $pred(v_5) \leftarrow pred(v_1) = null$. Obviously, the algorithm takes $O(\log n)$ rounds and is a BPPA. Moreover, a vertex v communicates with $pred(v)$ which may not be its direct neighbor.

Example 4: S-V Algorithm. [Yan et al., 2014b] adapts Shiloach-Vishkin’s PRAM algorithm for computing CCs [Shiloach and Vishkin, 1982] to work on Pregel, which is called the S-V algorithm. We present a simplified version of the S-V algorithm here which is more efficient. Throughout the algorithm, vertices are organized by a forest such that all vertices in a tree belong to the same CC. Each vertex v maintains a pointer $D[v]$ indicating its parent in the forest (i.e., $a(v) = D[v]$). We relax the tree definition a bit here to allow the tree root w to have a self-loop (i.e., $D[w] = w$).

At the beginning, each vertex v initializes $D[v] \leftarrow v$, forming a self loop as shown Figure 3.2(a). Then, the algorithm proceeds in rounds, and in each round, the pointers are updated in two steps: (1) *tree hooking* (see Figure 3.2(b)): for each edge (u, v) , if u ’s parent $w = D[u]$

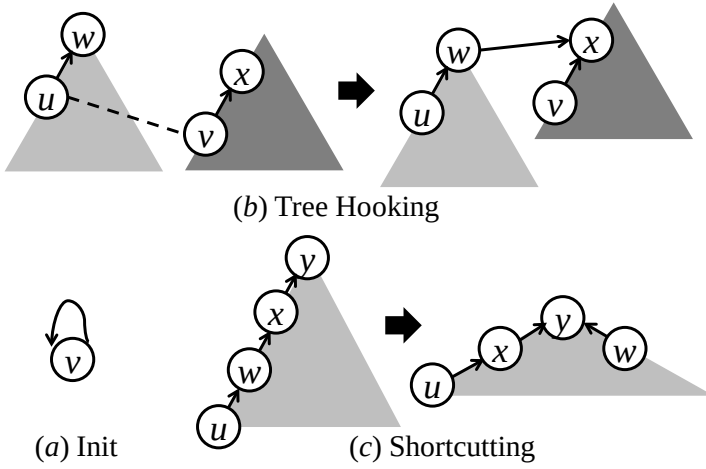


Figure 3.2: Illustration of the S-V Algorithm

is a tree root, we hook w as a child of v 's parent $D[v]$ (i.e., we merge the tree rooted at w into v 's tree); (2) *shortcutting* (see Figure 3.2(c)): for each vertex v , we move it closer to the tree root by pointing v to the parent of v 's parent, i.e., $D[D[v]]$. Note that Step 2 has no impact on $D[v]$ if v is a root or a child of a root.

The algorithm repeats these two steps until no vertex v has $D[v]$ updated in a round (checked by using aggregator), by which time every vertex is in a star (i.e., tree of height 1), and each star corresponds to a connected component. Since $D[v]$ monotonically decreases during the computation, at the end $D[v]$ equals the smallest vertex in v 's CC (which is also the root of v 's star). Similar to the request-respond operation in list ranking, each step of S-V can be formulated in Pregel as a constant number of supersteps, and since shortcutting guarantees the $O(\log |V|)$ -round bound, the algorithm is a PPA. However, it is not a BPPA since a parent vertex may communicate with many more vertices than its own neighbors (e.g., a root). Moreover, a vertex v communicates with $D[v]$, which may not be its direct neighbor.

3.3 Optimizations in Communication Mechanism

Since Google's Pregel is not public, a lot of open-source Pregel-like systems have been developed recently. The key feature of a Pregel-like system is that, it adopts a message passing model for programming and communication. This feature differentiates Pregel-like systems from those systems that adopt a shared memory abstraction. Existing Pregel-like systems improve the basic Pregel model from various aspects. This section introduce a few popular Pregel-like systems with improvements in the communication mechanism. We introduce more Pregel-like systems that improve the framework of Pregel from other aspects in the next three sections.

Giraph [Ching et al., 2015]. Apache Giraph is probably one of the most popular Pregel-like systems, which is written in Java and built on top of Hadoop. The good performance and scalability of Giraph is mainly contributed by Facebook's improvements to an earlier Giraph version. There are three important improvements which we list as follows. Firstly, Facebook researchers improved each worker to support multithreading in order to achieve fine grain parallelism. Secondly, the old Giraph version maintains vertices, edges and messages as native Java objects, which consumes excessive memory and garbage collection time; Facebook researchers solved the problem by serializing the edges and messages into byte arrays to reduce the number of objects. Thirdly, a superstep splitting technique is developed to split a message-heavy superstep into several steps, so that the number of messages transmitted in each step does not exceed the memory size. Superstep splitting is only effective when a received message can be aggregated to a value by the receiver vertex, or more formally, *compute(.)* is distributive (e.g., message summation in PageRank computation).

Pregel+ [Yan et al., 2015]. Pregel+ is written in C/C++, and thus recycles memory in time and keeps the memory footprint small. Since the system has full control of memory usage, all vertices, edges and messages are stored as main-memory objects and there is no need of serialization (except for sending messages). Pregel+ further developed two techniques to reduce the number of messages as described below.

The first technique is to create mirrors of each high-degree vertex v on all other workers that contain v 's neighbor(s). The adjacency list of v is partitioned among its mirrors, where each mirror maintains the sub-list of v 's neighbors in its local worker. In the PageRank (resp. Hash-Min) algorithm, v broadcasts the same value $a(v)/d_{out}(v)$ (resp. $a(v)$) to all its out-neighbors (resp. neighbors); for such an algorithm, v simply sends the value to every mirror, which then forwards the value to all (and possibly many) local neighbors. The message value towards a neighbor w (on worker W_w) may also be post-processed by the mirror of v on W_w using the edge value of (v, w) (according to an optionally user-defined function) before forwarding it to w . Since each worker has at most one mirror for v , the total number of messages incurred by v is bounded by the number of workers $|\mathbb{W}|$, which can be much smaller than v 's degree.

However, since a mirrored vertex forwards its value directly to its mirrors, it loses the chance of message combining. Therefore, there is a tradeoff between vertex mirroring and message combining in reducing the number of messages, and we should only mirror high-degree vertices. [Yan et al., 2015] proves that the number of messages is minimized when we mirror all vertices with degree at least $|\mathbb{W}| \cdot \exp\{d_{avg}/|\mathbb{W}|\}$ where d_{avg} is the average vertex degree.

The second technique is designed for pointer jumping algorithms where a vertex needs to communicate with a large number of other vertices that may not be its neighbors. To see this, consider the last round of the S-V algorithm described in Section 3.2, where a root r vertex needs to communicate with all other vertices v in r 's component, since they form a star rooted at r (i.e., $D[v] = r$). Here, each vertex v will send requests to r for the value of $D[r]$ in a superstep, and r will receive these requests and send $D[r]$ to every requesting vertex in the next superstep. Pregel+ prevents r from receiving and sending a lot of messages, by combining all requests on each worker as one request towards r , and r only responds to every requesting worker rather than every requesting vertex. To support this optimization, Pregel+ allows a vertex v to request for the value of another vertex (e.g., r), and the value (e.g., $D[r]$) can then be directly accessed in the next superstep.

GPS [Salihoglu and Widom, 2013]. GPS is written in Java, and can be more efficient than Giraph when a parameter called polling time is set as small value such as 10 ms (the default setting is 1 s, which forces any superstep to take at least 1 s) [Lu et al., 2014]. There are two main optimizations over the basic model of Pregel. Firstly, it supports a technique called LALP which is similar to vertex mirroring in Pregel+. However, to deal with the conflict between vertex mirroring and message combining, it simply does not perform sender-side message combining at all, and is thus less effective than Pregel+ in terms of message reduction. Secondly, since the workload may change during computation, GPS considers vertex migration for dynamic load balancing. However, vertex migration is too costly to be effective, as we shall discuss in Section 3.4.

MOCgraph [Zhou et al., 2014]. MOCgraph is developed on top of Giraph, and adopts the *message online computing* (MOC) model to eliminate the memory space consumed by messages. The idea is to let in-memory vertices absorb incoming messages directly without buffering them. While MOCgraph adopts asynchronous execution to allow faster convergence, it retains the concept of “superstep” which differentiates it from GraphLab. In each superstep of the MOC model, every vertex v calls a UDF *sendMessages(.)* exactly once to send messages, whose values are computed from v ’s latest value; meanwhile, each message received by a vertex u reads and updates u ’s latest vertex value by calling another UDF *onlineCompute(.)*. Note that this essentially requires that *onlineCompute(.)* is distributive (like message combiner of Pregel). Therefore, the MOC model generates and consumes messages in the same superstep, carrying no messages across supersteps.

To support synchronous execution like in Pregel, the UDF *v.sendMessages(.)* sends messages according to v ’s old value updated by the last superstep, rather than using the latest value. In other words, each vertex v maintains two values, the old value for computing outgoing messages, and the latest value to be updated by incoming messages.

If the memories cannot hold all vertices, MOCgraph partitions vertices into partitions on disks, and incoming messages have to be buffered to disks. Consider a message m targeted at vertex v in a par-

tition P . If P is not in memory, m has to be appended to a message file that corresponds to P . To process a vertex partition P , messages in the corresponding message file will first be loaded to update P , before vertices in P send messages. Note that keeping more vertices in memory allows more incoming messages to be absorbed without buffering, and thus MOCgraph separates edges from vertex partitions to allow more vertex partitions to be kept in memory (while only those edges of the current partition in processing are kept in memory). A hot-aware re-partitioning strategy is used to keep those partitions that tend to receive more messages in memory.

3.4 Load Balancing

In this section, we review some Pregel-like systems that improve load balancing during computation by two techniques: vertex migration and dynamic concurrency control.

Vertex Migration. A Pregel job may exhibit different workload distributions in different supersteps. For example, in Hash-Min, while all vertices send messages at the first superstep, few vertices are active in the last few supersteps. The set of active vertices (or more strictly, the set of vertices that send messages) in a superstep is called a working window, or simply, *wind*, by [Shang and Yu, 2013]. The idea of *vertex migration*, or *dynamic graph partitioning*, is to migrate vertices from workers with heavy workloads to those with light workloads during the computation, which also endeavors to reduce the communication cost (e.g., crossing machine edges).

However, there are two challenges in vertex migration: (1) migrating a vertex also requires migrating its adjacency list, which is more costly than sending a message; (2) it is difficult to *catch the wind*: for example, a migrated vertex may just converge and need no more computation, and a worker currently with a light workload may be heavily loaded in the next superstep due to the activation of many of its vertices. Even worse, since the vertex-to-worker relationship changes, we cannot simply derive the worker of a vertex by hashing its ID, and more costly method should be used. Note that the vertex-to-worker

mapping is critical to system performance, because when a vertex u sends a message to another vertex v , it needs to know which machine the message should be sent to (i.e., the machine that v resides in). Therefore, existing efforts on vertex migration turn out to be not very effective, which we review next.

[Shang and Yu, 2013] developed a Pregel-like system on top of HAMA, which keeps track of the vertex-to-worker mapping by a Lookup Table [Tatarowicz et al., 2012]. A few policies are designed based on the immediate previous *wind* to catch the *wind* in the next superstep. The best reported result is a 31.5% reduction of execution time for PageRank computation, while the ratio is merely 2% and 9% for shortest path computation and maximal matching.

Mizan [Khayyat et al., 2013] identifies the cause of workload imbalance using distributed measurements of the performance characteristics of all vertices, and constructs a vertex migration plan without requiring centralized coordination. The vertex-to-worker mapping is maintained by a distributed hash table (DHT). [Khayyat et al., 2013] partitioned vertices using hash-based ID partitioning, range-based ID partitioning and METIS [Karypis and Kumar, 1998] in their experiments, and found that for both hash-based and METIS partitioning, dynamic migration did not improve the results; while around 40% improvement was observed for range-based partitioning. Recently, [Han et al., 2014a] reported that Mizan does not function correctly with dynamic migration.

GPS also supports dynamic graph partitioning. Instead of using a lookup table to keep the vertex-to-worker mapping, GPS relabels the IDs of the migrated vertices so that the mapping can still be computed by the hash function. The additional overhead is to update the old ID in the adjacency lists of other vertices with the relabeled ID. During vertex migration, GPS keeps the number of vertices in each worker unchanged, and a vertex is migrated only if the number of messages is significantly decreased after the migration (controlled by a threshold). However, even the developers of GPS themselves do not recommend to enable vertex migration. For example, Semih said the following comments:

“... unless your job is going to run a large number of supersteps, running dynamic repartitioning will slow down your job ...”¹

“In general, I advice not to do dynamic repartitioning and also not to work on it. I think it’s very difficult to get benefits out of it in a real system implementation. The overheads are just too high.”²

Dynamic Concurrency Control. While previous explorations on vertex migration turned out not very effective, there is an existing work that dynamically balances the workload within each individual worker, which exhibits reasonable performance improvement. Specifically, [Shao et al., 2015] observed that a high-quality graph partitioning (e.g., from METIS) sometimes even decreases the overall performance in existing big graph systems, despite the reduced number of crossing-machine edges. This is because, while the number of messages from remote machines is reduced, the number of messages sent by local vertices significantly increases. However, existing systems are not partition-aware, and still allocate the same amount of computing resources (e.g., threads) to process remote messages and local messages, respectively. Therefore, the increased volume of local messages leads to a performance bottleneck (since the number of threads processing them is not increased), and the cost of processing local messages dominates the overall cost.

[Shao et al., 2015] developed the PAGE system, which adopts a *dynamic concurrency control* model, to overcome the above limitation. Each worker in PAGE monitors measurements such as message generation speed, local message processing speed and remote message processing speed. Based on these measurements, PAGE dynamically adjusts the numbers of threads for processing local and remote messages, respectively, so that (1) the speed of message processing matches the speed of incoming messages, and that (2) the numbers of threads assigned to process *local* and *remote* messages, are proportional to the incoming speeds of *local* and *remote* messages, respectively.

¹<https://groups.google.com/forum/#!searchin/stanfordgpsusers/repartition/stanfordgpsusers/3Wzlnm1eXbw/xf8c9hBUCGMJ>

²<https://groups.google.com/forum/#!searchin/stanfordgpsusers/repartition/stanfordgpsusers/HV04gc-2Tcs/T-tlgC2GpdgJ>

3.5 Out-Of-Core Execution

While most distributed big graph systems are in-memory systems, out-of-core support has recently attracted a lot of attention due to the real demands from academic institutes and small businesses that could not afford memory-rich clusters. For example, [Bu et al., 2014] reported that in the Giraph user mailing list there are 26 cases (among 350 in total) of out-of-memory related issues from March 2013 to March 2014. As another example, [Zhou et al., 2014] reported that to process a graph dataset that takes only 28GB disk space, Giraph and GraphLab need 370GB and 800GB memory space, respectively; and when memory resources become exhausted, the performance of Giraph degrades seriously while GraphLab simply crashes.

Two solutions are currently available for processing a big graph when memory space is insufficient. The first solution is to use a single-PC disk-based (or SSD-based) graph systems like GraphChi [Kyrola et al., 2012] and X-Stream [Roy et al., 2013]. Since these systems adopt the shared memory abstraction, we will review them in Chapter 5. However, the performance of these systems is limited by the disk bandwidth of one PC, and thus the processing time scales linearly with the graph size. To scale to larger graphs, the second solution is to use a distributed graph system that supports efficient out-of-core execution. In such a system, the graph is partitioned among all machines in a cluster, and during computation, each machine only processes its own part of the graph on the local disk. As a result, the bandwidth of all disks in a cluster is fully utilized, but the tradeoff is that the overhead of network communication is incurred.

We now briefly review the existing distributed big graph systems that support out-of-core execution. Giraph has been extended with out-of-core capabilities to solve the out-of-memory issues³, where users can enable “out-of-core graph” (resp. “out-of-core messages”) to store graph partitions (resp. “messages”) to local disk(s) if the in-memory buffer overflows. Users may set the maximum number of partitions and/or messages allowed to be kept in memory, and provide a list of

³<http://giraph.apache.org/ooc.html>

paths corresponding to different disks on each machine, so that Graph will access all disks in a round-robin fashion to fully utilize the disk bandwidth. Pregelix [Bu et al., 2014] models the semantics of Pregel by relational operations like join and group-by, and leverages a general-purpose dataflow engine for out-of-core execution. In contrast, GraphD [Yan et al., 2016d] tailors its out-of-core execution design to the computation model of Pregel, and is thus able to avoid expensive operations like join and group-by. Recently, Chaos [Roy et al., 2015] extends X-Stream to work in a distributed environment, but it is only efficient when network bandwidth far outstrips storage bandwidth (which is also an assumption in its system design).

In the next subsection, we review the GraphD system. Pregelix will be reviewed in Section 8.2 when we discuss dataflow-based systems, and GraphChi, X-Stream and Chaos will be reviewed in Sections 5.2.1 and 5.2.2 when we discuss systems that adopt shared memory abstraction.

3.5.1 GraphD

GraphD [Yan et al., 2016d] is designed to run on a cluster of commodity PCs connected by Gigabit Ethernet, which are readily available in academic institutes and small businesses. In this setting, the bandwidth of sequential disk scan is usually much higher than the actual network bandwidth [Yan et al., 2016d, Shen et al., 2014]. This is because (1) sequential disk scan is significantly accelerated by memory cache, and (2) the network resource is contended by all the workers in a cluster, limiting the point-to-point connection throughput. Therefore, if each worker streams disk-resident data (e.g., edges and messages) in parallel with message transmission, it is possible to hide the time for disk streaming inside the time for network communication. GraphD exploits this characteristic and achieves performance comparable to (and sometimes even faster than) an in-memory Pregel-like system.

Distributed Semi-Streaming Model. Like Pregel, GraphD partitions vertices among the workers in a cluster using a hash function *hash(.)*. However, GraphD adopts a novel *distributed semi-streaming* (DSS) computation model, where each worker only keeps the states of

its vertices in main memory, and the adjacency lists are stored in a file on local disk, which we call as the edge stream, denoted by S^E . The state of a vertex v includes its ID, $a(v)$, $active(v)$, and $d(v)$ (or $d_{out}(v)$ for a directed graph), whose memory space is bounded by a small constant.

A worker W organizes its in-memory vertex states by an array A , whose elements are ordered by vertex ID. The adjacency lists in S^E are also ordered by the ID of their corresponding vertices. In a superstep, the $compute(.)$ function is called on the active vertices in A in serial. Since a vertex v needs to access $\Gamma(v)$ in $v.compute(.)$, the next $d(v)$ items are sequentially read from S^E to form $\Gamma(v)$.

To skip the adjacency lists of a series of inactive vertices efficiently, W sums the degree of these vertices as sum , and forwards the read position in S^E by sum items. If data at the new position is not contained in the in-memory streaming buffer of S^E , the buffer is refilled with data starting from the new position. This design avoids reading all items in S^E when only a small fraction of vertices are active.

Since only vertex states are kept in memory, the total RAM space required by GraphD is only $O(|V|)$. Moreover, [Yan et al., 2016d] proved that in a cluster of workers, $\mathbb{W}$, every worker requires only $O(|V|/|\mathbb{W}|)$ RAM space for the state array A , with a high probability of $(1 - O(1/|V|))$. The RAM space required by streaming buffers and buffers for sending and receiving messages are negligible compared with A .

Message Streams. Like edges, messages are also treated as disk streams in GraphD. For simplicity, we only discuss Pregel algorithms where message combiner is applicable. Each worker W maintains $|\mathbb{W}|$ outgoing message streams $S_1^O, \dots, S_{|\mathbb{W}|}^O$, where each stream S_i^O stores messages towards the i -th worker, denoted by W_i . If a vertex on W sends a message to another vertex on W_i , the message is simply appended to S_i^O .

In order to perform vertex-centric computation (which generates messages) and sending generated messages in parallel, a message stream S_i^O is split into multiple files as follows: if the size of the current file written by S_i^O reaches a threshold $\mathcal{B}$, a new file is created for appending more messages. A sending thread probes all the $|\mathbb{W}|$ message streams,