SPRINGER BRIEFS IN COMPUTER SCIENCE

Da Yan
Yuanyuan Tian
James Cheng

Systems for Big Graph Analytics

Springer

SpringerBriefs in Computer Science

Da Yan • Yuanyuan Tian • James Cheng

Systems for Big Graph Analytics

Springer

Da Yan
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL, USA

Yuanyuan Tian
IBM Almaden Research Center
San Jose, CA, USA

James Cheng
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

ISSN 2191-5768 ISSN 2191-5776 (electronic)
SpringerBriefs in Computer Science
ISBN 978-3-319-58216-0 ISBN 978-3-319-58217-7 (eBook)
DOI 10.1007/978-3-319-58217-7

Library of Congress Control Number: 2017941194

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

Chapter 1
Introduction

Due to the growing need of processing big graph datasets (e.g., online social networks, knowledge graphs) created by modern applications, recent years have witnessed a surging interest in developing big graph analytics systems. Tens of systems have been developed for processing big graphs. Although they enrich the tools available for users to analyze big graphs, it is difficult for beginners of this field to gather up the threads of various system features and optimization techniques, and to choose the right system for their problems at hand.

As an effort to help readers obtain a comprehensive idea of big graph systems research, we have presented a tutorial in SIGMOD 2016,[1] and published a survey [1] that summarizes the novel ideas and technical contributions of various systems. This book serves slightly different purposes from the above works:

- Instead of a comprehensive coverage, this book selects a few systems that we consider the most important in the recent big graph systems research, for beginners to quickly get familiar with this area with minimal time investment.
- This book provides some tutorials on how to use important graph analytics systems, which could be helpful to beginners who would like to (1) use a graph analytics system, or (2) to get hands-on experience of developing Big Data systems, such as how to design a user-friendly and customizable API, how to send objects across machines, and how to interact with a distributed file system. The tutorials are of special interest to researchers who would like to learn how to build their own Big Data systems from scratch.

The book also shares some common features with our previous survey [1]:

- Except for the system tutorials, the descriptions are more from an academic perspective. In other words, we focus on novel ideas of system design (e.g., user-friendly API, effective performance optimizations) rather than the imple-

[1]http://sigmod2016.org/sigmod_tutorial2.shtml.

mentation details of individual systems. The book is also highly selective on academic values of the systems to be discussed.

- Our focus is on the computation models of graph analytics systems, rather than the data storage. As a result, we do not touch upon graph databases such as TITAN[2] despite their importance. Computation and storage are rather independent topics. In fact, even TITAN supports data ETL to Apache Giraph, a computation engine that we shall cover. We remark that unlike SQL queries in relational databases, the queries on graphs are highly heterogeneous and a one-size-fits-all query model is not very promising (e.g., consider random walks vs. graph matching).
- In addition to describing existing big graph systems research, we also provide our vision on future research directions in Chap. 8.

The book is organized into three parts. Part I introduces the idea of think-like-a-vertex programming that has been widely accepted in big graph systems research, and reviews the various vertex-centric systems. Part II goes beyond the vertex's perspective, and reviews two different kinds of frameworks that advocate the idea of think-like-a-graph programming. This new computation model has raised a lot of attention recently, and has been applied to either accelerate vertex-centric computation, or solve computation-intensive graph mining problems that cannot be well solved by vertex-centric systems. Part III introduces the matrix solution to big graph analytics. We also summarize the book and discuss the future directions at the end of this book.

Reference

1. D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1–2):1–195, 2017.

[2]http://titan.thinkaurelius.com/.

Part I
Think Like a Vertex

Chapter 2
Pregel-Like Systems

2.1 Google's Pregel

Many graph algorithms are **iterative** in nature. For example, the PageRanks of all vertices in a graph can be computed in iterations: in each iteration, every vertex sums the values received from its in-neighbors, updates its own PageRank using the sum, and then distributes the new PageRank among its out-neighbors [1]. The process continues until the PageRanks of all vertices converge, or a specified maximum number of iterations is reached.

Pregel [16] is designed for distributed in-memory execution of such iterative graph algorithms. It was a pioneering work since before Pregel's advent, people were performing iterative computation of big graphs using MapReduce [11, 14], a computation model ill-suited for iterative processing. Specifically, each MapReduce job can only implement one iteration of computation, and hence the huge amount of intermediate data between two consecutive iterations have to be dumped to and then loaded back from a **distributed file system (DFS)**, causing high disk and network IO overhead.[1]

This section first reviews the computation model of Pregel, and then introduces how to develop Pregel algorithms with performance guarantees. We begin by introducing the graph notations used throughout this book.

Graph Notations 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

[1]Writing to a DFS generates network overhead since each data block are replicated to multiple machines to tolerate machine failures.

D. Yan et al., *Systems for Big Graph Analytics*, SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-58217-7_2

v by $d(v)$. If G is directed, we denote the set of in-neighbors (resp. out-neighbors) of v by $\Gamma_{in}(v)$ (resp. $\Gamma_{out}(v)$), and denote the in-degree (resp. out-degree) of v by $d_{in}(v)$ (resp. $d_{out}(v)$). The graph diameter is denoted by δ. □

2.1.1 Computation Model

A Pregel job starts by loading an input graph G from a DFS into the main memories of all machines in a cluster, where vertices are partitioned among the machines. Typically, vertices are partitioned by a hash function *hash*(.) shared by all machines: each vertex v is assigned to a machine $M = hash(v)$. Each vertex v maintains its adjacency list which usually tracks its neighbors (e.g., $\Gamma(v)$, $\Gamma_{out}(v)$, or $\Gamma_{in}(v)$), so that v can communicate with these neighbors. A vertex v also maintains a vertex value $a(v)$, and 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 function (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 the DFS.

We can see that a Pregel job only interact with an underlying DFS at the beginning and end of its execution: vertex states (e.g., adjacency lists and vertex values) are pinned in memory, and messages generated in a superstep are exchanged among machines (for use in the next superstep) without going through the DFS.

Another key contribution of Pregel is that, it pioneered the idea of **vertex-centric programming** which is dominantly adopted by subsequent big graph systems. The vertex-centric model enables user-friendly programming: in order to develop a distributed graph algorithm, one only needs to specify the computation logic (like a state machine) for a generic vertex. This is essentially an SIMD (Single Instruction Multiple Data) programming model.

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. However, note that Pregel's API does not restrict a vertex to only communicate with its neighbors: a vertex v can communicate with any other vertex whose ID is known by v. This makes Pregel a very expressive computation model, and most subsequent systems assume only neighborhood communication, a more restricted model that enables more optimizations.

Example 2.1 (PageRank) We review the PageRank algorithm of [16] 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 2.2 (Hash-Min) We consider the Hash-Min algorithm of [31] for computing the connected components (CCs) of an undirected graph G. Given a CC, C, we denote the set of vertices of C by $V(C)$. Hash-Min labels each vertex v with the smallest vertex ID in v's CC: $cc(v) = \min\{id(u) : u \in V(C)\}$. 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. The algorithm converges in $O(\delta)$ supersteps, since the smallest ID in a CC propagates by one hop after each superstep. □

Note that these algorithms exhibit different performance behaviors. In the PageRank algorithm, all vertices participate in the computation in every superstep; while in Hash-Min, fewer and fewer vertices broadcast ID-messages as the computation proceeds, since $a(v)$ of more and more vertices converge to $cc(v)$.

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 machine M targeted at v_t will be combined into a single message by M 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 machine. The aggregated values are then aggregated globally at the master. Finally, the globally aggregated value is broadcast back to all machines for use in the next superstep.

Fault Tolerance To be fault tolerant, a Pregel job may be specified to periodically back up the state of computation to the underlying DFS at superstep boundaries as a checkpoint (e.g., every ten supersteps). Since data on a DFS is replicated on multiple machines, a checkpoint is resilient to machine failures. If a failure happens, all machines 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 (a) other vertices or (b) the edges of other vertices. Graph Mutations are useful in some graph algorithms, such as the algorithm for k-core finding [17] 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.

2.1.2 Algorithm Design

There are a number of papers studying Pregel algorithm design: Quick et al. [17] demonstrated that many social network analytic tasks can be formulated as Pregel algorithms, while Salihoglu and Widom [21] 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. For example, in the triangle finding algorithm of [21], 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 (since Pregel buffers messages in memory to be processed by *compute*(.)).

Users need to be aware of the scalability of a Pregel algorithm, and to design their algorithm to be scalable. For example, the previous triangle finding algorithm may run for k rounds where each round i sends inquiries for only the subset of vertices satisfying $id(v)$ modulo k equals $(i-1)$. The adapted algorithm reduces memory consumption and supports effective checkpointing. It is found in [31] that many fundamental graph problems have scalable Pregel algorithms, and they can be used as building blocks to construct scalable graph algorithms for other graph problems. We now introduce this class of scalable algorithms.

Scalable Pregel Algorithms 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 a Pregel algorithm with a large number of supersteps is inefficient since each superstep requires a global barrier at the end, which incurs round-trip network delay (assuming that the reliable transport-layer protocol TCP is adopted).

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). We call such a Pregel algorithm simply as a **practical Pregel algorithm** (**PPA**).

The Hash-Min algorithm presented in Sect. 2.1.1 is not a PPA, since it takes $O(\delta)$ supersteps; if the whole graph is just a long path and the vertex with the smallest ID is on one end of the path, Hash-Min runs for $|E|$ supersteps. 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 CCs [31], the algorithm of [31] for computing biconnected components, and the algorithm of [21] for computing minimum spanning forest. We now illustrate the idea of applying pointer jumping in Pregel by looking at two PPAs.

Example 2.3 (List Ranking) List ranking is an important step of the Euler tour technique and finds wide applications such as computing biconnected components [31]. The problem is defined as follows. We assume elements in a list are linked backwards from the tail to the head (the algorithm for forward linking is similar). Consider a linked list L with n vertices (or items), where each vertex v is associated with a value $val(v)$ and a link to its predecessor $pred(v)$. The vertex v at the head of L has $pred(v) = null$. For each $v \in L$, 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 $v \in L$. If $val(v) = 1$ for any $v \in L$ (see Fig. 2.1), then $sum(v)$ is simply the rank of v in the list, i.e., the number of vertices preceding v plus 1. Note that vertices of the input data are stored in a DFS in arbitrary order.

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 2.1 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

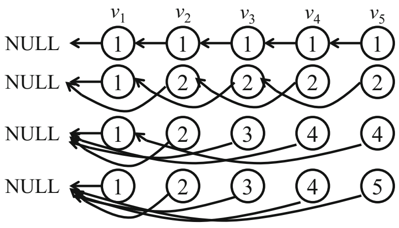


Fig. 2.1 Illustration of the BPPA for list ranking

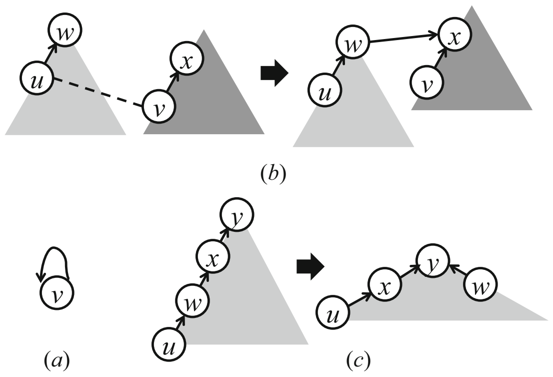


Fig. 2.2 Illustration of the S-V algorithm. (**a**) Init, (**b**) tree hooking, (**c**) shortcutting

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 $pred(v_5) \leftarrow pred(v_1) = null$. The algorithm takes $O(\log n)$ rounds since the run-length of summation is doubled (if available) after each round. Moreover, a vertex v communicates with $pred(v)$ which may not be its direct neighbor.

Example 2.4 (S-V Algorithm) Shiloach–Vishkin's PRAM algorithm for computing CCs [26] is adapted by Yan et al. [31] to work in 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 Fig. 2.2a. Then, the algorithm proceeds in rounds, and in each round, the pointers are updated in two steps: (1) *tree hooking* (see Fig. 2.2b): for each edge (u, v), if u's parent $w = D[u]$ is a tree root, and $D[v] < D[u]$ (i.e., $x < w$ as compared by vertex ID), 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 Fig. 2.2c): for each vertex v, we move it closer to the tree root by pointing v to the parent of v's parent, i.e., $D[D[v]]$. It is easy to see 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.

2.2 Pregel-Like Systems

In this section, we introduce the various off-the-shelf open-source Pregel-like systems, so that readers can get familiar with their features and be able to choose a proper one to use according to their needs. There are two critical aspects to consider when choosing a Pregel-like system: (1) system efficiency and the supported optimizations, (2) software robustness which is often decided by the amount of users and the technical group supporting the system.

At the early stage, as Google did not open source Pregel, other companies in need of a vertex-centric graph system started to build one of their own. The collaborative effort gave rise to an open-source system called **Giraph**,[2] similar to the effort of open-source Google File System [8] and MapReduce [6] over 10 years ago which created Hadoop.[3] Giraph was initiated as a side project at Yahoo!, and made efficient and popular due to improvements by Facebook researchers. Giraph was written in Java, and thus naturally integrates with Hadoop. It is probably the most popular Pregel-like system for the time being, with project team members spanning Facebook, LinkedIn, Twitter, Pivotal, HortonWorks, etc. While there exists some minor optimizations to the original model of Pregel, the main goal of Giraph is to open-source Pregel.

Another system, **GraphX** [10], provides vertex-centric API for graph processing on top of Spark [34], a general-purpose big data system. The benefit of using GraphX is that, for a complex task where graph processing is only part of it, the whole task can be written in Spark to avoid data import/export overhead between graph and non-graph jobs, hence improving the overall end-to-end performance.

For graph processing alone, we recommend Giraph over GraphX, since we found that GraphX requires significantly more memory space which translates to renting more machines in a cloud service, and GraphX is also slower since its underlying

[2]http://giraph.apache.org/.

[3]http://hadoop.apache.org/.

engine is a dataflow engine not dedicatedly designed for vertex-centric graph processing. Also, Spark's advantage is mainly enjoyed by operations with "narrow dependency", such as condition-based filtering of data items, where a partition of filtered data can be directly generated in a machine without communicating with other machines, and if the partition is lost, it can be recovered from the data partition that it filters from. Pregel's vertex-wise message passing has "wide dependency", where network communication dominates the cost and discounts the benefit of memory caching of data. Fault tolerance in GraphX also reverts back to checkpointing, since a lost data partition (e.g., messages received by a set of vertices) is often contributed by many (if not all) previous data partitions, which has to be recovered first.

We focus here more on the **academic perspective**, i.e., novel technical contributions (e.g., optimization techniques) on top of Pregel's original model. Numerous systems have been developed to optimize Pregel's simple model from various aspects including communication mechanism, load balancing, out-of-core support, fault recovery, and on-demand querying. While some systems (often just prototypes) from the academia are not as robust as those from the industry (if ever open-sourced) due to lack of testing and small number of users, there are good systems with high performance such as **GraphLab** [9, 15] (now commercialized as the company Turi[4] and focuses more on machine learning), and the graph processing platform **BigGraph@CUHK**.[5]

Both GraphLab and BigGraph@CUHK were implemented in C++. In contrast, Giraph (resp. GraphX) was implemented in Java (resp. Scala) that runs on top of JVM. Unfortunately, Java is ill-suited to hold a large number of objects (e.g., vertices) in memory due to the overhead of GC (Garbage Collection) and a higher per-object memory footprint [3]. In fact, the old Giraph version maintains vertices, edges and messages as native Java objects, which consume excessive memory and garbage collection overhead; Facebook researchers solved the problem by serializing the edges and messages into byte arrays to reduce the number of objects [5]. We remark that while Java (and Scala) allows system developers to be more productive (important for commercial purposes which play an important part in influencing the Big Data market), C++ permits higher efficiency. The programming simplicity of both C++ and Java solutions are the same as their end-users program with the same vertex-centric API adopted from Pregel.

The rest of this section is organized according to the optimization aspects of existing Pregel-like systems, which are more from the academic perspective and provides a big picture of the various possibilities beyond the basic model of Pregel.

[4]https://turi.com/.

[5]http://www.cse.cuhk.edu.hk/systems/graph/.

2.2.1 Communication Mechanism

In Sect. 2.1.1, we described two algorithms for computing PageRanks and connected components, whose *compute*(.) is distributive and message combiner is applicable; in Sect. 2.1.2, we presented the S-V algorithm where a parent vertex v needs to respond its attribute to all requesting children vertices (that are not necessarily direct neighbors), and message combiner is inapplicable. Existing optimization techniques are mostly designed for the former type of Pregel algorithms, while Pregel+ [29] provides separate techniques for both types of algorithms.

Giraph [5] is improved by Facebook from its earlier version. The improvements including multithreading support, vertex serialization (mentioned before), and a *superstep splitting* technique to reduce the memory space consumed by buffering messages. Superstep splitting is only applicable to Pregel algorithms whose *compute*(.) is distributive. The technique splits a message-heavy superstep into several steps, so that the number of messages transmitted in each step is reduced. The message values received by each vertex are directly aggregated (i.e., combined but at receiver side) and there is no need to buffer them. MOCgraph [35] eliminates the memory space consumed by messages, through a more aggressive *message online computing* (MOC) model that lets in-memory vertices absorb incoming messages directly without buffering them.

Pregel+ [29] developed two techniques to reduce the number of messages. The first technique is designed for combiner-applicable algorithms, and creates mirrors of each high-degree vertex v on all other machines that contain v's neighbor(s). The adjacency list of v is partitioned among its mirrors on different machines. To see how vertex mirroring reduces communication, consider the Hash-Min algorithm: when a high-degree v broadcasts $a(v)$ to all its neighbors, it only needs to send $a(v)$ to every mirror, which then forwards the value to all local neighbors. The message value may also be post-processed by a mirror using the edge value in local adjacency list. Since each machine has at most one mirror for v, the total number of messages sent by v is bounded by the number of machines 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 sender-side message combining, and thus low-degree vertices should not be mirrored. Yan et al. [29] prove a degree threshold to guide vertex mirroring that minimizes the number of messages. GPS [20] provides a similar technique called LALP but provides no analysis on the tradeoff with message combining. In fact, it simply does not perform sender-side message combining at all.

Pregel+'s second technique extends the basic API of Pregel with a request-respond functionality, which is designed for pointer jumping algorithms like S-V where a vertex needs to communicate with a large number of other vertices that may not be its neighbors. For example, consider later rounds of S-V, where a root vertex r needs to communicate with many other vertices v in r's component with $D[v] = r$. Here, each vertex v will send requests to r for $D[r]$ in a superstep, and r will receive these requests and respond $D[r]$ back in the next superstep. The request-respond API allows v to pose requests to r, and each machine combines all local requests to r as one request that gets sent to *r*; *r* only responds to every requesting machine rather than every requesting vertex, and *v* may look up *D*[*r*] from the table of received responses in the next superstep.

2.2.2 Load Balancing

There are two potential load balancing issues with Pregel. The first issue is the **straggler problem**: the slowest machine is assigned many high-degree vertices and thus sends many more messages than average, and since Pregel adopts synchronous execution, the performance is decided by the slowest machine. The straggler problem can be generated by the uneven vertex degree distribution as is often the case for real-world power-law graphs, and it can also be generated due to workload changes across different supersteps (e.g., in Hash-Min).

A solution to this problem is **vertex migration**, i.e., to migrate some vertices of a heavily-loaded machine to a lightly-loaded machine. There are three challenges with vertex migration: (1) migrating a vertex *v* means migrating everything related to *v* including its adjacency list, which is costly; (2) it is difficult to changing workload: for example, a migrated vertex may just converge and need no more computation, and a currently lightly-loaded machine may be heavily loaded in the next superstep due to many vertices being activated; (3) the vertex-to-machine relationship changes and cannot simply be tracked by a function *hash*(.) (for sending messages to a target machine), and more costly method is needed.

Currently, WindCatch [23], Mizan [12] and GPS [20] all support vertex migration. WindCatch and Mizan tracks the vertex-to-machine mapping through a distributed lookup table, while GPS relabels the IDs of the migrated vertices so that the mapping can still be computed by *hash*(.). However, the performance improvement is very limited (if not exacerbated by additional overheads) and even GPS's developer Semih recommended not to enable vertex migration and even to do research on it (see Chap. 3.4 of [28] for more discussions).

The second issue of load balancing is with the computation and communication workload distribution within each individual machine, and the solution is by dynamic concurrency control. Specifically, Shao et al. [24] observed that a high-quality graph partitioning (e.g., from METIS) sometimes even slows down the overall performance in Giraph, 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, and the number of threads receiving and processing local messages becomes insufficient. PAGE [24] adds dynamic concurrency control support to Giraph, to overcome the above limitation. Each machine 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, to balance the incoming speeds of *local* and *remote* messages, respectively, and to balance the speed of message processing and the speed of incoming messages.

2.2.3 Out-of-Core Execution

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. [2] 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; Zhou et al. [35] reported that to process a graph dataset that takes only 28 GB disk space, Giraph needs 370 GB memory space due to the need of storing intermediate data such as messages.

There are solutions that use a single-PC disk-based (or SSD-based) graph systems like GraphChi [13] and X-Stream [19] and GridGraph [36]. However, the performance of these systems is limited by the disk bandwidth of one PC, and thus the processing time increases linearly with the graph size. To scale to larger graphs, distributed out-of-core systems emerged including Pregelix [2], GraphD [33] and Chaos [18]. Since a graph is partitioned among all machines in a cluster, and each machine only processes its own part of the graph on the local disk, the bandwidth of all disks in a cluster is fully utilized, but the tradeoff is the overhead of network communication.

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 issues,[6] where users can configure to store graph partitions (resp. "messages") to local disk(s) if the in-memory buffer overflows. Pregelix [2] models Pregel's API with relational operations like join and group-by, and leverages a general-purpose dataflow engine for out-of-core execution. In contrast, GraphD [33] 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. Chaos [18] 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).

We now introduce GraphD [33] in greater detail. GraphD 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 much higher than the actual network bandwidth [25, 33], and GraphD exploits this characteristic to hide the time of streaming data (e.g., edges and messages) on disks inside the time of message transmission, achieving performance comparable to an in-memory Pregel-like system.

In each superstep, each machine streams the edges of its assigned vertices. To allow efficient sparse access when only a small portion of vertices are active, the streaming algorithm skips k edges if a series of vertices are inactive (whose sum of degree equals k): if the next edge at the new position is not cached in the in-memory streaming buffer, the buffer is refilled with data starting from the new position. This design avoids reading all edges when only a small fraction of vertices are active.

[6]http://giraph.apache.org/ooc.html.

Like edges, messages are also treated as disk streams. For simplicity, we only discuss the design for Pregel algorithms where message combiner is applicable, and we show that only one pass of the incoming and outgoing message streams is necessary with $O(|V|)$ memory usage. Specifically, every machine maintains an outgoing message stream to store messages targeting each machine; a message stream is organized as a series of size-constrained files so that messages can be simultaneously fetched from the head-file (that is cached in memory) and appended to the tail-file (a new file is created if tail-file size overflows). At each time, a message sending thread loads a head-file from an available message stream, and performs in-memory message combining (requiring no more memory asymptotically than the number of vertices in the target machine). Meanwhile, a message receiving thread keeps receiving each message block, and aggregates the contained messages to its vertices' partially aggregated messages in memory. It is not difficult to see that $O(|V|)$ memory is sufficient and each superstep requires only one pass over disk streams.

2.2.4 Fault Recovery

Fault recovery aims to allow a long-running job to survive (or recover quickly from) the crash of any machine, rather than to restart computation from the very beginning. For this purpose, during normal execution, the state of computation should be periodically backed up to a resilient storage (e.g., an underlying DFS) which incurs additional overhead during failure-free execution. If a failure happens, computation can revert back to the latest backed-up state to continue with recovery.

In Sect. 2.1.1, we saw that Pregel supports fault tolerance by **checkpointing**, i.e., writing the current vertex values, edges and messages to the DFS, and recovery is executed by letting all machines reload the state of computation from the latest checkpointed superstep. This baseline approach is not efficient enough, and optimization techniques have been proposed to improve the performance of checkpointing and recovery, including lightweight checkpointing, recovery by message-logging, and their hybrid, which we describe next.

We first look at **lightweight checkpointing** as proposed in [32]. A checkpoint of Pregel contains (1) vertex states which take $O(|V|)$ space, (2) adjacency lists which take $O(|E|)$ space, and (3) all messages generated in the superstep. The messages usually take $O(|E|)$ space (e.g., in PageRank computation [16]), but can be much larger (e.g., $O(|E|^{3/2})$ in triangle counting [17]). Noticing that the adjacency lists are static in most Pregel algorithms, Yan et al. [32] adopt incremental checkpointing to avoid rewriting edges that do not change from the previous checkpoint. For Pregel algorithms without topology mutations, adjacency lists only need to be saved in the first checkpoint to be loaded for later recovery; for deletion-only Pregel algorithms like the k-core finding algorithm of [17], the amount of checkpointed edge updates is totally $O(|E|)$ rather than $O(|E|)$ with each checkpoint.

Messages are also removed from the checkpoint of [32] (and hence the name *lightweight checkpoint*), and are directly generated from checkpointed vertex states during recovery. This is straightforward for some algorithms like PageRank, where v simply sends messages $a(v)/d_{out}(v)$ (computed from the checkpointed $a(v)$) to every out-neighbor; while some minor reformulation is required for other algorithms like Hash-Min: to include a flag in $a(v)$ indicating whether v received a smaller ID, and during recovery, only broadcast v's value if the flag is set. See [32] for the adapted API that supports lightweight checkpointing for a general Pregel algorithm.

While lightweight checkpointing reduces the checkpointing overhead during failure-free execution, **message-logging based recovery** [25] aims to quickly recover from a failure when it actually happens. We illustrate the idea by considering the following scenario. Assume that a Pregel job fails at superstep s_{fail}, and the latest checkpoint is at superstep s_{cp}. Then, all vertices roll their states back to superstep s_{cp} and then restarts the normal execution. This approach wastes computation since the state of a vertex surviving the failure is already at superstep s_{fail} and there is no need to recompute it from superstep s_{cp}. The only use of recomputation is to feed messages to reloaded vertices, as their recomputation needs these messages.

Since Shen et al. [25] find that local disk streaming is much faster than message transmission in a Gigabit Ethernet environment, Shen et al. [25] proposed to let each vertex log the messages that it generates (in *compute*(.)) to the local disk before sending them. The local message logging adds negligible overhead since concurrently executed message transmission is slower. During recovery, only those messages that target at the reloaded vertices need to be transmitted. Specifically, a surviving vertex simply forwards its logged messages towards those reloaded vertices, rather than executing *compute*(.) to generate messages. The recovery is much faster as the communication workload is much lower than during normal execution. A recovery algorithm that is robust to cascading failures is non-trivial and see [25] for the detailed algorithm.

Message-logging slows down failure-free execution, since when a new checkpoint is written, previously logged messages need to be deleted to avoid using up disk space, which is expensive since the logged messages span multiple supersteps all the way back to the previously checkpointed superstep. The problem is solved by Yan et al. [32], which only logs vertex states rather than messages, significantly reducing the amount of logged data. During recovery, messages are generated from the logged vertex states, and then forwarded to the reloaded vertices.

The approaches seen so far below to *coordinated checkpointing* [7], and for asynchronous execution models like that of GraphLab [9, 15], *uncoordinated checkpointing* can be applied such as Chandy-Lamport snapshot [4]. Among other methods for fault tolerance, optimistic recovery [22] eliminates the need of checkpointing at all, for a narrower class of self-correcting fix-point algorithms, since the converged results are the same no matter how the values of failed vertices are re-initiated. Imitator [27] avoids checkpointing by constructing k replicas of each vertex on k different machines. As long as less than k machines crash, each vertex still contains at least one replica and computation is not lost. See Sect. 3.6 of [28] for more discussions on fault recovery methods.

2.2.5 On-Demand Querying

Most Pregel-like systems are designed for offline graph analytics, where a job visits most (if not all) vertices in a big graph for many iterations. However, there is also a need to answer graph queries online, where each query usually accesses only a small fraction of vertices during the whole period of evaluation. However, offline analytics systems cannot support query processing efficiently, nor do they provide a user-friendly programming interface to do so, which we explain below.

Weaknesses of Offline Analytics Systems If we write a vertex-centric algorithm for a generic query, we have to run an independent job for each incoming query. In this solution, each superstep transmits only the few messages of one light-workload query and cannot fully utilize the network bandwidth. Also, there are a lot of synchronization barriers, one for each superstep of each query. Moreover, some systems such as Giraph bind graph loading with graph computation for each job, and the loading time can significantly degrade the performance.

An alternative solution is to hard-code a vertex-centric algorithm to process a batch of k queries, where k can be an input argument. However, in *compute*(.), one has to differentiate the incoming messages and/or aggregators of different queries and update k vertex values accordingly. In addition, existing vertex-centric framework checks the stop condition for the whole job, and users need to take care of additional details such as when a vertex can be deactivated (e.g., when it should be halted for all the k queries), which should originally be handled by the system itself. Last but not least, this approach does not solve the problem of low utilization of network bandwidth, since in the later stage, only a small number of straggler queries are still being processed.

Quegel [30] The Quegel system is designed to efficiently answer light-workload graph queries on demand. A user only needs to specify the Pregel-like algorithm for a generic query, and Quegel processes graph queries posed online by effectively utilizing the cluster resources. Quegel also provides a convenient interface for constructing indexes to expedite query evaluation, which are not supported elsewhere.

Quegel adopts a **superstep-sharing** execution model. Specifically, Quegel processes graph queries in iterations called *super-rounds*. In a super-round, every query that is currently being processed proceeds its computation by one superstep; while from the perspective of an individual query, Quegel processes it superstep by superstep as in Pregel. Intuitively, a super-round in Quegel is like *many queries sharing the same superstep*. For a query q whose computation takes n_q supersteps, Quegel processes it in $(n_q + 1)$ super-rounds, where the last super-round prints the results of q on the console or dumps them to HDFS (Hadoop Distributed File System).

In Quegel, new incoming queries are appended to a query queue (in the master), and at the beginning of a super-round, Quegel fetches as these queued queries to start their processing. During the computation of a super-round, different machines run in parallel, while each machine processes (its part of) the evaluation of the queries

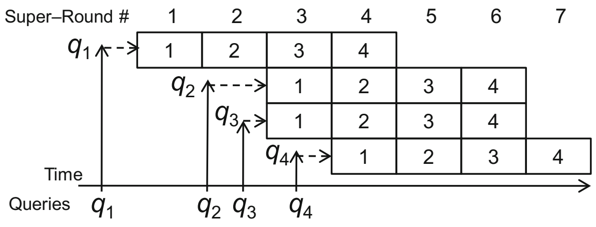


Fig. 2.3 Illustration of superstep-sharing

in sequence. If a query q finishes its evaluation and the result gets reported/dumped, the resources consumed by q are then released.

For the processing of each query, the supersteps are numbered. Two queries that enter the system in different super-rounds have different superstep number in any super-round. Messages (and aggregators) of all queries are synchronized together at the end of a super-round, to be used by the next super-round. Figure 2.3 illustrates the execution of four queries q_1, q_2, q_3 and q_4 in our superstep-sharing model, where we assume for simplicity that every query takes four supersteps.

In terms of system design, Quegel manages three kinds of data: (1) *V-data*, whose value only depends on a vertex v, such as v's adjacency list; (2) *VQ-data*, whose value depends on both a vertex v and a query q, including $a(v)$, $active(v)$, and v's incoming message queue; (3) *Q-data*, whose value only depends on a query q, such as query content, superstep number, aggregator, and control information. Q-data are maintained by every machine, and kept consistent at the beginning of each superstep. Each vertex maintains its V-data, and a table of VQ-data for queries in processing.

A Quegel user writes vertex-centric programs exactly like in Pregel, and the processing of concrete queries is transparent to users. For example, a user may access $a(v)$ (resp. superstep number) in $v.\,compute(.)$, and if a machine is processing v for query q, the VQ-data of v and q (resp. the Q-data of q) is actually accessed. To be space efficient, a vertex v allocates a state (i.e., VQ-data) for a query q only if q accesses v during its processing. Specifically, when vertex v is activated for the first time during the processing of q, the VQ-data of q is initialized and inserted into the VQ-data table of v. After a query q reports or dumps its results, the VQ-data of q is removed from the VQ-data table of every vertex that has been accessed by q.

Quegel also provides a convenient API for users to build distributed graph indices. After a machine loads its assigned vertices in memory, it may build a local index from its vertices using the user-defined logic, before Quegel starts to process queries. For example, in graph matching (or graph keyword search), an inverted index can be built that maps each label (or keyword) to the local vertices that contain the label (or keyword), so that vertices containing the required labels (or keywords) can be activated to start graph traversal.

References

1. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World-Wide Web Conference (WWW)*, pages 107–117, 1998.
2. Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.
3. Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 2013.
4. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
5. A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.
6. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
7. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.
8. S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.
9. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
10. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.
11. U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.
12. Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.
13. A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.
14. J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *MLG*, pages 78–85. ACM, 2010.
15. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
16. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.
17. L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463, 2012.
18. A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.
19. A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.
20. S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.
21. S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.
22. S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. "All roads lead to Rome": optimistic recovery for distributed iterative data processing. In *CIKM*, pages 1919–1928, 2013.
23. Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, pages 553–564, 2013.
24. Y. Shao, B. Cui, and L. Ma. PAGE: A partition aware engine for parallel graph computation. *IEEE Trans. Knowl. Data Eng.*, 27(2):518–530, 2015.
25. Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast failure recovery in distributed graph processing systems. *PVLDB*, 8(4):437–448, 2014.

26. Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.
27. P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *DSN*, pages 562–573, 2014.
28. D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1–2):1–195, 2017.
29. D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.
30. D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *PVLDB*, 9(7):564–575, 2016.
31. D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.
32. D. Yan, J. Cheng, and F. Yang. Lightweight fault tolerance in large-scale distributed graph processing. *CoRR*, abs/1601.06496, 2016.
33. D. Yan, Y. Huang, J. Cheng, and H. Wu. Efficient processing of very large graphs in a small cluster. *CoRR*, abs/1601.05590, 2016.
34. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.
35. C. Zhou, J. Gao, B. Sun, and J. X. Yu. Mocgraph: Scalable distributed graph processing using message online computing. *PVLDB*, 8(4):377–388, 2014.
36. X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.

Chapter 3
Hands-On Experiences

3.1 Why Beginning with BigGraph@CUHK

There are a few open-source Pregel-like systems that users can choose from, the most popular of which is Giraph [2]. We would recommend users to begin with Giraph (for Java programmers) or our BigGraph@CUHK toolkit (for C++ programmers); other systems may have not been extensively tested and lack documentation,[1] or may be designed more for compatibility with general dataflow engines than for the efficiency of graph computation (e.g., GraphX [5]).

This chapter provides a step-by-step tutorial on how to use BigGraph@CUHK, with some notes on how the systems of BigGraph@CUHK are designed. It is aimed for readers who would like to get their hands dirty using a Pregel-like system and/or learning how to develop their own Big Data systems, and you may skip this chapter if hands-on experience is not your focus.

We adopt BigGraph@CUHK for the tutorial due to the following reasons:

- The systems of BigGraph@CUHK are neatly designed. The core system architecture is much simpler than other Pregel-like systems, allowing users to learn quickly and to flexibly make changes by adding new functionalities and optimizations, or even reframing into a new Big Data system. BigGraph@CUHK serves as a shortcut to help researchers focus on new contributions to computation models and optimization techniques themselves, relieving them from the cumbersomeness of understanding the heavy code of a poorly-documented open-source system (or reinventing the wheel). Research productivity can be significantly boosted by building your work on top of BigGraph@CUHK. In fact, our systems have led to a number of our high-quality publications in the field of vertex-centric

[1] GPS [10] stops computation once all vertices are inactive, even when there is on-the-fly messages; Mizan [8] is reported to return incorrect results when vertex migration is enabled [7].

distributed graph processing [9, 11–20], and have been used by other researchers such as in [3].

- BigGraph@CUHK exposes key elements in Big Data system design to users, such as how to interact with Hadoop Distributed File System (HDFS) (using C/C++), how to transmit data across machines (by wrapping MPI functions), why and how to (de)serialize data, how to stream data on external memory, etc., which help users better understand the basic principles behind the scene of Big Data systems. Such key elements are often automated and hidden from users in other Big Data systems (unless users are willing to read their heavy code). In this sense, BigGraph@CUHK is a better option for education and academic use.
- BigGraph@CUHK is highly efficient as it is built upon C++. While higher-level languages such as Java, Python and Scala have become quite popular these days due to higher programming productivity in the industry, the execution speed is of no match to C++ implementation. Moreover, automatic garbage collection of higher-level languages only creates new problems for **in-memory** Big Data processing. In fact, even though Hadoop MapReduce always streams data from/to disks and networks, it has to adopt *object reuse*[2] to control the number of objects during the in-memory key-value processing by mappers and reducers. Giraph maintains the large number of vertices as in-memory objects for iterative computation, which leads to poor performance over large graphs. The problem is fixed by Facebook researchers who also resort to object reuse [2]. As another example, GraphX [5] easily consumes an order of magnitude more memory than other Pregel-like systems and yet exhibits poor performance [1, 30]. C++ is actually a more convenient choice for in-memory processing, as long as system developers take care of garbage collection properly.

It may not be totally fair to claim that Pregel-like systems using C++ are more difficult to use than those using other languages, since end users see the same user-friendly vertex-centric API. One may at most claim that the job of system development itself is more difficult for developers not familiar with C++. After all, C++ based graph processing systems such as GraphLab [4] and BigGraph@CUHK gain popularity due to their efficiency and user-friendliness. The only weakness of using a C++ based parallel system is that debugging becomes more difficult if a segment fault happens and one has to insert some assertions to assist debugging; in contrast, higher-level languages will throw exceptions so that the error can be exactly located. In other cases (e.g., when execution gets stuck), debugging is standard like inserting some isolation and printing statements (debugging a parallel program is never easy).

This chapter assumes that readers have certain familiarity with C++ programming and Linux. However, we utilize HDFS (which is implemented in Java) for

[2]Object reuse means an object is deserialized from a binary stream for processing, after which the updated object is serialized back to the binary stream. As an example of object reuse, the key and value objects to the *reduce*(.) function are reused as indicated in the API documentation: https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapred/Reducer.html.

distributed data storage to avoid reinventing the wheel.[3] To us, it is sufficient to abstract HDFS as (1) a logical file system whose disk space aggregates the disk space of all machines, where (2) each file is stored in the unit of 64 MB data blocks, and each data block is replicated on three different machines[4] (so that data is not lost even if two machines are down), and where (3) data blocks of a big file are concurrently read (or written) by all machines to utilize their aggregate disk bandwidth.[5] We remark that almost all open-source distributed Big Data systems support data loading (resp. dumping) from (resp. to) HDFS, to enjoy the benefits of aggregate disk bandwidth, fault tolerance, and an abstract file system view, and the difference often lies in the computation model on top.

3.2 System Deployment and Running

BigGraph@CUHK can be accessed from http://www.cse.cuhk.edu.hk/systems/graph/.[6] It consists of five systems that improve or extend the basic framework of Pregel from various aspects: Pregel+, Blogel, Quegel, GraphD, and LWCP. We focus on Pregel+[7] in this section, since the design and deployment of the other systems are similar.

Software Dependency Pregel+ was implemented in C/C++ as a group of header files, and users only need to include the necessary base classes and implement the application logic in their subclasses. Pregel+ interacts with HDFS through libhdfs, a JNI-based C API for HDFS. We use the term "worker" to represent the basic computing unit that performs serial execution, and a machine may run multiple workers. In Pregel+, each worker is simply an MPI (Message Passing Interface) process and communications among different workers are implemented using MPI's communication primitives. Pregel+ should be deployed in Linux, and compiled with G++.

According to the above description, we need to make sure that every machine is deployed with (1) G++, (2) MPICH or OpenMPI (i.e., an implementation of the MPI standard),[8] (3) JDK (for running Hadoop), and (4) Hadoop (where we only use HDFS). Oftentimes, a cluster is already deployed with one or more of these software, but we still need to configure the cluster (e.g., environmental variable) so that a Pregel+ program can be compiled with libhdfs. While Pregel+ works with any

[3]Hadoop's JNI-based C API, called libhdfs, supports HDFS reads/writes using C.

[4]We assume the default replication factor 3, which can be otherwise configured; if data is uploaded from a machine inside the HDFS cluster, one replica must be on that machine.

[5]As a simplified example, if a 100-node cluster needs to load a 100 GB data, each machine actually only needs to read 1 GB data which is executed in parallel with other machines' reads.

[6]You can also search "BigGraph@CUHK" in Google.

[7]http://www.cse.cuhk.edu.hk/pregelplus/.

[8]According to our experience, MPICH provides a more stable performance while OpenMPI often makes the first few supersteps in Pregel+ running for longer than normal before stabilizing.

version of GCC (GNU Compiler Collection), MPI and JDK, we do require users to choose the proper version of system code that matches your Hadoop version, i.e., Hadoop 1.x or Hadoop 2.x (YARN), due to their different libhdfs API.[9]

Introduction to libhdfs libhdfs is a bridge between HDFS (the ideal distributed storage platform for Big Data) and efficient C/C++ processing of the data stored in HDFS. It is directly included in any Hadoop release. For example, in Hadoop 2.6.5,[10] under folder "include" we can find the C header file "hdfs.h" that lists all the functions of libhdfs's API, where each function is preceded with comments that explain what is the function doing and how to call it; while library files of libhdfs can be found under "lib/native". These two paths need to be specified in the makefile in order to compile a Pregel+ program.

As an example of libhdfs's C function, consider the following function:

```
int hdfsDelete(hdfsFS fs, const char* path, int recursive);
```

The comments above indicates that the function is used to delete file specified by "path" in the file system specified by "fs", and the third parameter "recursive" is explained as

```
* @param recursive if path is a directory and set to
* non-zero, the directory is deleted else throws an exception. In
* case of a file the recursive argument is irrelevant.
```

In other words, if *path* specifies a folder, and *recursive* is specified as a non-zero integer (e.g., 1), all files and folders under *path* will be recursively deleted, which is useful since a big file on HDFS is often stored as many moderate-sized files under a data folder so that these files can be independently read and written in parallel. In general, to process a file in HDFS, one needs to first get the file system handle of type *hdfsFS*, by connecting to the proper host and port of HDFS (e.g., using *hdfsConnect*(.)), and then get the file handle of type *hdfsFile* (e.g., using *hdfsOpenFile*(.)), and all other IO operations then use these two handles, which should be garbage-collected at last (e.g., using *hdfsCloseFile*(.) and *hdfsDisconnect*(.)). A complete introduction of libhdfs's API is out of the scope of this book, and we refer readers to the header file "hdfs.h" for the detailed usage.

It is worth noting that libhdfs changes with Hadoop's version, and is not completely backward compatible. The directory structure of Hadoop also changes with Hadoop's version. For example, in Hadoop 1.2.1,[11] header file "hdfs.h" is under "src/c++/libhdfs", while library files of libhdfs for 64-bit OS are under "/c++/Linux-amd64-64/lib", and those for 32-bit OS are under "/c++/Linux-i386-32/lib". Also, *hdfsDelete*(.) in "hdfs.h" now has the form:

```
int hdfsDelete(hdfsFS fs, const char* path);
```

[9]Hadoop generations are not fully backward compatibility. For the time being, Hadoop 3.x is already available in alpha versions but has not been tested with BigGraph@CUHK; however, we remark that BigGraph@CUHK only requires a stable version of HDFS, and even Hadoop 1.x works perfectly well and there is not much difference in terms of disk IO performance anyway.

[10]http://apache.mirrors.pair.com/hadoop/common/hadoop-2.6.5/hadoop-2.6.5.tar.gz.

[11]http://archive.apache.org/dist/hadoop/core/hadoop-1.2.1/hadoop-1.2.1.tar.gz.

This function is different from *hdfsDelete*(.) in Hadoop 2.6.5 since the last input argument *recursive* does not exist. This leads to code written with libhdfs of Hadoop 1.x failing to compile with libhdfs of Hadoop 2.x. Theoretically, it is easy to realize backward compatibility in Hadoop 2.x using default argument like "*recursive* = 1", but unfortunately libhdfs is a C API but default argument is a feature of C++. Other differences exist, e.g., Hadoop 2.x recommends to get file system handle using *hdfsBuilderConnect*(.) rather than *hdfsConnect*(.) which is deprecated. Due to these reasons, Pregel+ maintains two versions of HDFS-related code for each system, one for Hadoop 1.x and the other for Hadoop 2.x.

Also note that libhdfs only provides a binary data interface (comparable to Hadoop's Java package "org.apache.hadoop.fs"), and there is no text interface comparable to, for example, Hadoop MapReduce's LineRecordReader. Therefore, Pregel+ implements a wrapping header file that exposes a text interface to HDFS over the binary data interface of libhdfs.

Notes on Deployment Pregel+'s website provides a detailed *Documentation* webpage,[12] where the first two links provide foolproof tutorials for deploying Pregel+ in a multi-node cluster with Hadoop 1.x and Hadoop 2.x, respectively. A few more words might be helpful before you follow the tutorials to deploy Pregel+, which are given as follows.

The tutorials assume you have access to root privilege, and a bare OS, and thus MPICH and Hadoop are installed under "/usr/local" (rather than a user's home folder "/home/user_account", or simply "∼") to be shared by all users. Of course, a user needs to add MPICH and Hadoop to the environment variables in "∼/.bashrc", and add himself as an owner of Hadoop's home folder using the *chown* or *setfacl* command.

However, the assumption of root privilege may not be true in a public cluster, in which case you have to install software (e.g., MPICH and Hadoop) under your home folder "/home/your_account" (or simply "∼") rather than "/usr/local". In this case, the relevant paths mentioned in the tutorials should be changed accordingly, and there is no need to use "sudo" or "su -c", and to add ownership to Hadoop's home folder. If you are just trying Pregel+ in a standalone environment, Hadoop should be deployed in pseudo-distributed mode instead.

Several points are worth noting for the deployment. (1) It is important to configure your user account to be able to have password-less SSH connection from master to all slave machines, which is required by both MPI and Hadoop. This also makes it convenient to copy files (e.g., configured software, compiled MPI programs) to slave machines using the *scp* command. (2) To allow Pregel+ code to recognize libhdfs, it is important to add the library path of libhdfs to environment variable *LD_LIBRARY_PATH*, and to add the relevant jar files in Hadoop's home folder to environment variable *CLASSPATH*. Now, you are ready to make Pregel+ deployed by following[13] our online tutorial!

[12]http://www.cse.cuhk.edu.hk/pregelplus/documentation.html.

[13]Remember to make proper changes according to your setting as we just described!

Notes on Running a Pregel+ Program The third link[14] in Pregel+'s *Documentation* webpage provides a foolproof tutorial for compiling and running a Pregel+ program. Now that you have deployed Pregel+ on your cluster, it is time to test how it works by following that tutorial.

A few more words might be helpful. Firstly, any system in BigGraph@CUHK are actually just a folder of C++ header files,[15] and to use them, you just need to include the folder when compiling (e.g., in the makefile), with "-I[*path of the system code's folder*]". This is equivalent to incorporate all code under the system folder as part of your user program. In other words, the previously mentioned deployment procedure applies to all systems in BigGraph@CUHK: to use a system, download the system folder and put it anywhere you like, and then include the folder path in the makefile if you want to compile a program with that system. You can even update a system's code as you wish, e.g., by adding some printing statements to the header files to assist debugging; it suffices to include your system version in the makefile to compile. This is actually a plus of using BigGraph@CUHK: you can not only insert debugging statements in your user code, but also in system code.

Since a Pregel+ program is essentially an MPI program that reads and writes HDFS, *mpic++* is used as the command for compiling, and *mpiexec* is used as the command for running. Typically, a user compiles his code into a Pregel+ program on the master, and then uses *scp* to copy the program to all slaves for execution.[16] Since libhdfs does not provide an interface like Java API's InputSplit for delineate line records in a big file, all systems in BigGraph@CUHK require that (1) a large file be stored as a set of small files to be distributed among workers for parallel processing, and that (2) a line cannot span multiple files. Therefore, one cannot upload a big local file to HDFS simply using the command *hadoop fs -put*, but rather using the program provided at the end of the tutorial. Suppose the compiled program is named as *put*, then one can upload a big file simply as follows:

```
./put {local file} {HDFS path}
```

3.3 System Design and Basic Pregel API

Let us first look at Pregel+. The system and application code can be downloaded from Pregel+'s *Download* webpage.[17] Figure 3.1 shows the structure of Pregel+'s system code, where the second layer consists of four folders (libraries) of header files, and we only show important header files for the third layer.

Among the libraries, (1) "utils" implements the basic functionalities used by upper-level systems in BigGraph@CUHK, such as global information of a

[14]http://www.cse.cuhk.edu.hk/pregelplus/console.html.

[15]We directly write implementation in header files for simplicity.

[16]Programs should be under the same path as required by MPICH for running.

[17]http://www.cse.cuhk.edu.hk/pregelplus/download.html.

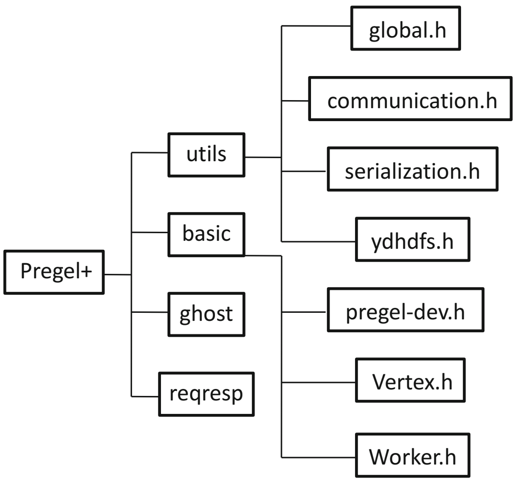


Fig. 3.1 Pregel+ system code structure

worker, text interface to HDFS, and tools for data serialization and transmission; (2) "basic" implements the original computation model of Pregel; (3) the other two libraries implements the Pregel variants with techniques to reduce the number of messages. Let us first focus on "utils" and "basic", which are representative of BigGraph@CUHK's system design.

3.3.1 The "utils" Library

Data Serialization In a distributed system, we cannot directly send an in-memory object to another machine, unless we indicate how it can be serialized into (and deserialize from) a binary stream. This is because an object o may contain fields that reference to other objects dynamically allocated in the heap, and thus o's data is not consecutive in memory.

In Java-based systems such as Hadoop and Giraph, serialization is achieved through the *Writable* interface,[18] which requires a user to indicate how to serialize (resp. deserialize) an object by specifying the abstract method $write(out)$ (resp. $readFields(in)$). In other words, to let vertices send integer messages, one should specify the message type to be *IntWritable* rather than *int* (or *Integer*).[19] User-defined type should implement these two functions (e.g., by recursing on all fields) to support (de)serialization. In a nutshell, one has to wrap a class into a new one that implements *Writable* to allow remote transmission.

[18] https://hadoop.apache.org/docs/r2.6.1/api/org/apache/hadoop/io/Writable.html.

[19] See https://cwiki.apache.org/confluence/display/GIRAPH/Shortest+Paths+Example for an example of how *Writable* types are used in Giraph.

The cumbersomeness of using a wrapper class is avoided in BigGraph@CUHK, thanks to C++'s support of operator overloading. To send an integer message, the message type can simply be *int*, as long as they support the following two functions that overloads the operators << and >>, respectively:

```
obinstream& operator>>(obinstream& m, T& data)
ibinstream& operator<<(ibinstream& m, const T& data)
```

Here, T is the type (e.g., basic data type, struct, or class) that should be serializable, *obinstream* (resp. *ibinstream*) is the type of a binary input (resp. output) stream[20] m. Recall that in C++, "cout<<" (resp. "cin>>") appends text data to the standard output (resp. input), and thus is comparable to *ibinstream* (resp. *obinstream*).

The header file "utils/serialization.h" defines *obinstream* and *ibinstream*, and the overloaded (de)serialization functions for basic types (e.g. *int*, *std::string*) and STL containers such as *std::vector<T>*. The serialization function is recursively defined. For example, we serialize an object o of type *std::vector<T>* to output stream m using $m << o.size()$, followed by $m << o[i]$ (assuming T is serializable) iterating index i over all elements in o. In general, we require that all template arguments of a class C to be specified with serializable types, so that C can be serialized by recursing on all fields of C. Users can directly use types with predefined (de)serialization functions, but more complex user-defined types need to implement the two functions that overload << and >> (e.g., recursively).

Line 16 in "utils/serialization.h" shows that *obinstream* maintains a byte array *buf* of type *vector<char>*, and serialized data are simply appended to the end of *buf*. Similarly, Line 152 shows that shows that *ibinstream* tracks a byte array *buf* and the next byte position in *buf* to deserialize data from, *index* (Line 154). Here, *ibinstream*'s constructor directly accepts a received byte array of known length, but garbage collection of the array is automatically performed in the destructor. In other words, one only needs to allocate space in heap to receive data, and to pass it to a *ibinstream* object o; o will automatically release the space after deserialization is done and o is no longer needed.

Data Transmission Data transmission is the process of sending data from one machine to another. It is what end users do not want to take care of, and thus, all Big Data systems try to hide communication details in their programming model. In contrast, from the system's perspective, data transmission is an indispensable component, and there are solutions such as *message passing interface* (MPI) used by BigGraph@CUHK, *remote procedure call* (RPC) used by GraphLab, Netty[21] used by Giraph, and ZeroMQ[22] used by Husky.[23]

[20]The name could be a bit confusing; here an input stream is named *obinstream* (resp. *ibinstream*) since the streamed data get sent to (resp. received from) another machine, and is thus outgoing (resp. incoming).

[21]https://netty.io/.

[22]http://zeromq.org/.

[23]http://www.cse.cuhk.edu.hk/proj-h/.

BigGraph@CUHK adopts MPI since it is a very mature technology that has been optimized and used for decades. MPI supports sending data of various types; for example, integer (resp. byte) can be specified by the type MPI_INT (resp. MPI_CHAR). However, recall that "utils/serialization.h" serializes all objects to send as byte streams (i.e., byte arrays), and thus, the sole MPI data type used by BigGraph@CUHK for transmission is MPI_CHAR.

The header file "utils/communication.h" wraps the low-level MPI communication primitives in byte level, into high-level data transmission operations in object level, so that users can send and receive objects of any type T. See Lines 49–96 for how the object-level transmissions are defined over binary-stream level transmissions, which is in turn defined over byte-level transmissions that directly corresponds to MPI primitives coping with data of type MPI_CHAR.

An important operation in Big Data systems is all-to-all communication, where every machine has data to send every other machine in a cluster. This is useful, say, for data shuffling in MapReduce, and for vertex-message exchange in Pregel.[24] However, MPI's all-to-all primitives[25] is notoriously known to be very slow when the number of participating processes is large. As a result, "utils/communication.h" implements its own all-to-all version using MPI's point-to-point sending and receiving primitives (e.g., Line 101). Other functionalities such as gathering (resp. broadcast) data from (resp. to) all workers directly call MPI's group communication primitives. These functions are in turn used, for example, in aggregator's processing. A detailed description of MPI and how they are used in "utils/communication.h" is out of the scope of this book, and we refer interested readers to Pregel+'s *Communication Primitives* webpage[26] for more details.

Text Interface to HDFS Recall that libhdfs only provides a binary interface to HDFS, i.e., one can only read (resp. write) a stream of bytes from (resp. to) HDFS. The header file "utils/ydhdfs.h"[27] builds a text interface on top of the binary interface of libhdfs, for use by systems of BigGraph@CUHK. Since libhdfs is not backward compatible, "utils/ydhdfs.h" actually redirects to "utils/ydhdfs1.h" (for Hadoop 1.x) or "utils/ydhdfs2.h" (for Hadoop 2.x, or YARN). To configure "utils/ydhdfs.h" for Hadoop 1.x (resp. YARN), comment (resp. uncomment) "#define YARN" at Line 4.

We briefly introduce some key classes and functions of our text interface, using "utils/ydhdfs1.h". The type *LineReader* at Line 80 defines a text input stream that reads a line at each time. The byte array *line* at Line 90 is used to keep the latest line read from a file on HDFS, whose capacity is automatically doubled if the current capacity is fully filled but the end of a line has not been reached during reading (e.g.,

[24]This is a simplified solution where messages to send should all be computed (often very quickly) before starting the all-to-all data exchange; a more efficient way is like in BigGraph@CUHK's GraphD system where messages can be sent in batches concurrently with computation.

[25]Here, the primitives refer to MPI_Alltoall and MPI_Alltoallv.

[26]http://www.cse.cuhk.edu.hk/pregelplus/communication.html.

[27]The name "ydhdfs.h" starts with "yd" because it is written by YAN, Da, the book's first author.

due to its large length). Internally, an in-memory byte array *buf* (at Line 82) is used to support batched reading: a file read is performed only when all cached bytes in *buf* are exhausted; and each time *readLine*() is called to read a line, the bytes in *buf* gets appended to *line* until the end of a line (i.e., '\n') is reached.

Similarly, *LineWriter* at Line 431 allows users to write content as a line at each time to a file on HDFS. It further cuts the output stream into files each about 8 MB, but guarantees that a line never crosses two files. If appending one more line to a file makes it exceed 8 MB, the file is closed and a new file is created for appending.

The *put*(.) function at Line 559 reads a local file line by line (using *LineReader*), and writes the lines to HDFS (using *LineWriter*). This function is used for putting a large file to HDFS, and the big file is split into a directory with many small files of size 8 MB, so that they can later be distributed to different machines for parallel reading. For example, the file assignment can be done by the function *dispatchRan*(*inDir*) at Line 937, which first sorts all files under directory *inDir* in non-increasing order of size, and then assign each file to the worker currently with the smallest cumulative file size. This method has an approximation ratio of 4/3 to the most load-balanced assignment [6].

BufferedWriter at Line 703 is used more often than *LineWriter* for workers to output the results of parallel computing to HDFS. This is because *BufferedWriter* requires users to output all texts including the end of a line '\n', which means that a Big Data system can pass a *BufferedWriter* object to end users in a user-defined function (UDF), where users can output as many lines as they want. Similarly, an in-memory buffer *buf* (at Line 708) is maintained for appending data, which only gets flushed to disk when becoming full.

Worker's Global Information Each MPI process runs the same piece of code, though on different data (e.g., disjoint sets of vertices). Therefore, the global information of a worker (i.e., an MPI process) can be treated as C++ global variables, which we store in "utils/global.h". Some important variables in the header file include: (1) _my_rank (at Line 19), which is the ID of the current worker, or equivalently, the rank[28] of the current process; (2) _num_workers (at Line 20), which is the total number of workers. They are initialized by *init_workers*() (at Line 30), which should be called at the beginning of any Pregel+ program.[29] Also, *worker_finalize*() (at Line 37) should be called at the end of any Pregel+ program.

Examples of other global variables include *global_combiner* and *global_agg regator*, which specifies the classes defining combiner and aggregator computation logics, respectively. These two variables have type *void* *,[30] which can be explicitly converted into any type through a cast operator. The system converts them into the proper types when using them.

[28]In MPI, if there are totally n processes, then they are numbered as $0, 1, \cdots, n-1$, and the number of a process is called its rank.

[29]This is because MPI_Init should be called at the beginning of an MPI program.

[30]This is equivalent to the *Object* class in Java, which is the base class of any other class. In fact, *Object* is a reference which can point to any object created in heap using the *new* operator, and is thus like a pointer which is *void* * here.

3.3.2 The "basic" Library

Let us assume that a graph is stored as a file (or a directory of files) on HDFS, where each line corresponds to a vertex v which stores information like v's ID and adjacency list. In any Pregel-like system, in addition to data types for vertices and messages, users need to specify three user-defined functions, respectively, for (1) translating a line into a vertex object, (2) vertex-centric computation logic, (3) writing information of a processed vertex to HDFS.

Java-based Pregel-like systems such as Giraph [2] and Pregelix [1] inherits Hadoop MapReduce's features like using independent input/output format classes to specify the data import/export format. For example, the demo example of Giraph[31] defines a vertex class *SimpleShortestPathsVertex* to specify the vertex-centric computation logic, but needs to define four other classes for data import/export: *SimpleShortestPathsVertexReader*, *SimpleShortestPathsVertexInputFormat*, *SimpleShortestPathsVertexWriter*, *SimpleShortestPathsVertexOutputFormat*, each of which needs to specify several generic types of their base classes properly. This is really a cumbersome design in order to reuse Hadoop classes in system development.[32]

All systems of BigGraph@CUHK simplify the user interface by requiring users to (1) specify data types and vertex-centric computation logic by subclassing a vertex base class, and to (2) specify data import/export by subclassing a worker base class which also used to start the distributed execution. In the context of Pregel+, users only need to subclass the *Vertex* and *Worker* classes (and possibly *Combiner* and *Aggregator* if combiner and aggregator are used).

The *Vertex* Class This vertex base class is defined in the header file "basic/Vertex.h". There are four template arguments for users to specify: (1) *KeyT* is the type of vertex ID, (2) *ValueT* is the type of vertex value, (3) *MessageT* is the message type, and (4) *HashT* is the type of an object that overloads the "()" operator to map a vertex ID to a worker ID. The default *HashT* is defined for *KeyT* being an integer (i.e., class *DefaultHash* at Line 12), but if *KeyT* is another type, *HashT* should be provided. The header file "utils/type.h" defines some pairs of *KeyT* and *HashT* that can be directly used, such as *intpair* at Line 14 and *IntPairHash* at Line 77. Users can define *HashT* for their own *KeyT* similarly.

A *Vertex* object maintains three fields: (1) vertex ID *id* (at Line 27), (2) vertex value *_value* (at Line 104), and (3) boolean variable *active* (at Line 105) indicating whether the vertex is active. We let users include adjacency list(s) in vertex value (i.e., type *ValueT*) rather than make it part of the base class, since this provides more

[31] https://cwiki.apache.org/confluence/display/GIRAPH/Shortest+Paths+Example.

[32] Hadoop provides pre-defined formatting classes like *KeyValueTextInputFormat* with simple parsing semantics, and field-to-field parsing is left to *map*(.) (for input) and *reduce*(.) (for output); but Giraph and Pregelix ask users to perform field-to-field vertex parsing by defining their own formatting subclasses, which makes user code very verbose.

flexibility. For example, users can customize each list item to just contain neighbor ID or to also include attributes like edge length, and users can maintain two lists, one for in-neighbors and one for out-neighbors. These fields can be updated by *Vertex*'s functions like *vote_to_halt*() at Line 88, and a *Vertex* object can also call function *send_message*(*id*, *msg*) to send a message to the specified vertex.

The *Vertex* class provides an abstract function *compute*(*messages*) for users to implement their vertex-centric algorithms. To write a vertex-centric program, a user only needs to subclass *Vertex* by specifying its template arguments and implement *compute*(*messages*), in which it can call functions like *vote_to_halt*() and *send_message*(.).

As an example, consider our example code for Hash-Min,[33] which consists of two files, "pregel_app_hashmin.h" which defines the user's *Vertex* subclass, and "run.cpp" which runs the vertex-centric program using the user-defined logic in "pregel_app_hashmin.h".

Let us focus on class *CCVertex_pregel* defined in Line 27 of "pregel_app_hashm in.h". It specifies *KeyT* and *MessageT* as *VertexID* which is essentially *int*,[34] and specifies *ValueT* as the type *CCValue_pregel* defined at Line 7 which includes the smallest ID seen *color*, and the adjacency list *edges*. We require *KeyT*, *ValueT* and *MessageT* to be serializable, which makes an vertex object serializable (see Lines 38–50 in "basic/Vertex.h"). This is because a worker that parses a vertex v may not be the worker that v is hashed to for computation, and thus vertex objects may need to be transmitted before the actual vertex-centric computation.

In Hash-Min, whenever a vertex v sends messages, it broadcast *color* to all its neighbors, and therefore Line 30 defines a *broadcast*(*msg*) function using *send_message*(.), which is then used in implementing *compute*(*messages*).

The *Worker* Class This vertex base class is defined in the header file "basic/Worker.h". It takes the user-defined *Vertex* subclass as an argument, and an optional aggregator class. It inherits the various types from the *Vertex* subclass (see Lines 19–21) which can be used in its two UDFs for data import/export:

```
virtual VertexT* toVertex(char* line)
virtual void toline(VertexT* v, BufferedWriter& writer)
```

The first function (at Line 268) defines how to parse a line into a vertex object, in which users only need the create a vertex object using the *new* operator, set its field properly and return it in pointer type. Using pointers avoids deep vertex copy in C++. Vertex deletion is automatically handled by Pregel+ (see Lines 61 and 87). The second function defines what to write to HDFS (using *writer*) for a processed vertex. This function is called for each vertex after the graph computation is done. Referring back to our Hash-Min example code, class *CCWorker_pregel* at Line 70 is the worker subclass that implements these import/export functions.

[33]http://www.cse.cuhk.edu.hk/pregelplus/code/apps/basic/hashmin.zip.

[34] *VertexID* is defined in Line 83 of "utils/global.h".

Worker also has a function *run*(*params*) (at Line 311) which defines the whole computation process, including vertex loading, vertex exchange (realized by *sync_graph*() defined at Line 71), vertex-centric iterative computation, and result dumping. To run a vertex-centric program, users only need to let his/her user-defined *Worker* subclass call *run*(*params*) with the input arguments *params* properly specified (contents include the data import/export path on HDFS), and every worker will start to run.

3.3.3 Summary

For Pregel+, all relevant header files are already included in "basic/pregel-dev.h", and thus a user program of Pregel+ only needs to include this header file. We remark that this is still a brief introduction of Pregel+'s system and API to help readers get started with using Pregel+. We have omitted other issues like defining your message combiner and aggregator, and system details like the message buffer implemented in "basic/MessageBuffer.h".

To write vertex-centric programs with Pregel+, please read the last two links of Pregel's *Documentation* page, which introduces the relevant architectural design[35] and the complete API.[36] If you want to revise or reuse the system (or parts of it), it would be helpful to read the system code, and the presentation in this chapter should be helpful to get you started.

All other systems of BigGraph@CUHK has a similar website structure and system design, and can be learned similarly as with Pregel+. It is worth noting that if your cluster is too small to perform in-memory computation, GraphD[37] provides a great alternative choice by streaming edges and messages on disks. The system code has a header file "ioser.h" that defines two classes *ofbinstream* and *ifbinstream* that stream data on local disks with the help of a small in-memory buffer. The system also allows vertex-centric computation to run concurrently with message transmission to hide disk IO cost, rather than to perform message transmission after vertex-centric computation as in the other systems.

References

1. Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.
2. A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[35] http://www.cse.cuhk.edu.hk/pregelplus/overview.html.

[36] http://www.cse.cuhk.edu.hk/pregelplus/api.html.

[37] http://www.cse.cuhk.edu.hk/systems/graphd/.

3. X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang. Computing connected components with linear communication cost in pregel-like systems. In *ICDE*, pages 85–96, 2016.
4. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
5. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.
6. R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.
7. M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.
8. Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.
9. Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.
10. S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.
11. H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 649–658, 2015.
12. D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1–2):1–195, 2017.
13. D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big graph analytics systems. In *SIGMOD*, pages 2241–2243, 2016.
14. D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.
15. D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.
16. D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *PVLDB*, 9(7):564–575, 2016.
17. D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.
18. D. Yan, J. Cheng, and F. Yang. Lightweight fault tolerance in large-scale distributed graph processing. *CoRR*, abs/1601.06496, 2016.
19. D. Yan, Y. Huang, J. Cheng, and H. Wu. Efficient processing of very large graphs in a small cluster. *CoRR*, abs/1601.05590, 2016.
20. Q. Zhang, D. Yan, and J. Cheng. Quegel: A general-purpose system for querying big graphs. In *SIGMOD*, pages 2189–2192, 2016.

Chapter 4
Shared Memory Abstraction

4.1 Programming Interface and Its Expressiveness

In a system that adopts shared memory programming abstraction, a vertex v directly accesses the data values of its adjacent vertices and edges, rather than passively receiving messages pushed by other vertices like in Pregel. The API restricts a vertex to only interact with its direct neighbors, since the underlying execution engine is often not shared memory. As a result, these systems are less expressive than Pregel, and cannot support algorithms that use pointer jumping such as those described in Sect. 2.1.2.

We denote the value of a vertex v by D_v, and the value of an edge (u, v) by $D_{(u,v)}$. A vertex v can access the following five types of data, where we regard u (and w) as an in-neighbor (and an out-neighbor) of v:

1. D_u for any $u \in \Gamma_{in}(v)$;
2. $D_{(u,v)}$ for any $u \in \Gamma_{in}(v)$;
3. D_v;
4. $D_{(v,w)}$ for any $w \in \Gamma_{out}(v)$;
5. D_w for any $w \in \Gamma_{out}(v)$.

We call the set consisting of these values as the *scope* (or *full-scope*) of vertex v. We also defined two other scopes: (1) *vertex-scope*, where a vertex v can only access D_u, D_v and D_w; and (2) *edge-scope*, where a vertex v can only access $D_{(u,v)}$, D_v and $D_{(v,w)}$. Full scope access is supported by GraphLab [2, 9], while the two more restricted scopes are adopted by single-PC disk-based systems that emerged later.

4.2 GraphLab and PowerGraph

A Brief History of GraphLab GraphLab pioneered the vertex-centric shared memory programming abstraction. It was originally developed to run in a single machine [8], and was extended to a distributed setting. Distributed GraphLab [9] keeps the shared memory programming abstraction of a single-machine environment, where a vertex can access its full scope, but the underlying engine is no longer shared memory. Its later version, PowerGraph [2], then overcame several weaknesses of distributed GraphLab, and changed the programming API as a side effect. GraphLab emphasized on machine learning applications since its birth. The GraphLab team created a startup called Dato, which was then renamed as Turi and acquired by Apple.

4.2.1 GraphLab

As a platform for machine learning, GraphLab adopts vertex scheduling to favor iterative algorithms where vertex values converge asymmetrically. Similar to *compute*(.) in Pregel, distributed GraphLab [9] requires a user to specify a UDF *update*() to be called by a vertex v. In v.*update*(), v may read and update the values in its scope, and submit any of the vertices in its scope to the scheduler for further processing.

Programming Model To illustrate how to write *update*(), we consider PageRank computation. In v.*update*(), v directly reads D_u of all $u \in \Gamma_{in}(v)$ to compute its new PageRank D_v^{new} and checks whether $|D_v^{new} - D_v|$ is larger than a convergence threshold ε. If not, D_v is considered as converged and there is nothing to do; otherwise, v updates $D_v \leftarrow D_v^{new}$ and adds all out-neighbors to the scheduler for further processing. This algorithm is much faster than the PageRank algorithm of Pregel, since more and more vertices have their PageRank values converged during the execution, and thus, fewer and fewer vertices call the *update*() function.

Arguably, one may simply vote a vertex v to halt in Pregel if v's PageRank changes by less than ε; both this method and that of GraphLab's may omit the transmission of some small values, leading to approximate results. However, the result quality is often acceptable and the significantly reduced computation workload is more attractive. In Sect. 4.2.3, we will see another asynchronous system that guarantees result exactness while still benefits from asymmetric convergence.

System Design GraphLab adopts asynchronous execution and hides network communication from its programming model. To see why network communication is inevitable, consider an edge (u, v), where u is assigned to a machine M_u while v is assigned to another machine M_v. Since both M_u and M_v need to maintain D_u, $D_{(u,v)}$ and D_v to allow u and v to directly access them, these overlapped data (termed *ghosts* in [9]) need to be synchronized across different machines.

This is in contrast to Pregel, where users explicitly send messages in their programs. However, the asynchronous model requires additional efforts to enforce data consistency under race conditions (e.g., by using locks). In fact, GraphLab has a synchronous mode that simulates the computation model of Pregel, and both [10] and [3] found that the synchronous mode is faster than the asynchronous mode for algorithms where asymmetric convergence does not help (e.g., Hash-Min).

GraphLab over-partitions a graph into many vertex partitions (called *atoms*) so that they can be evenly distributed to machines in a cluster of arbitrary size, avoiding the need of repartitioning. This vertex-partitioning approach has scalability issues as we analyze next. For each vertex partition, it is desirable to reduce the scope overlap with other partitions, since data in the scope overlap need to be replicated and synchronized. However, computing a high-quality partitioning (e.g., using ParMetis [5]) for a big graph is very expensive, while using vertex-ID hashing leads to a large amount of overlap. Specifically, a vertex u may have neighbors v spanning many different machines (if not all). The resulting high replication factor leads to huge memory consumption, and limits the system scalability. In [9], the largest graph tested has merely 200M edges, which is too small for distributed processing.

4.2.2 PowerGraph

GraphLab soon replaced its vertex-centric programming model by a GAS programming model which associates UDFs with the adjacent edges of each vertex, in a subsequent version called PowerGraph [2]. It is claimed in [2] that the new programming model allows edge partitioning among the machines, rather than vertex partitioning; this allows the edges of a high-degree vertex in a power-law graph to be processed concurrently by multiple machines, achieving better load balancing.

However, the improvement of load balancing is not likely to be very significant, since the huge number of vertices tend to offset the variance of vertex degree. We conjecture that another important reason is to reduce memory consumption due to data replication. Using the greedy edge assignment strategy to be described shortly, [2] is able to scale to a tested graph with 1.5B edges. Nevertheless, we remark that large memory consumption remains probably the biggest weakness of GraphLab, a price paid to simulate shared memory abstraction in a distributed setting.

The GAS Model PowerGraph adopts a Gather-Apply-Scatter (GAS) model for both programming and computation. Put simply, a vertex v first gathers values along adjacent edges, and aggregates these values to compute its vertex value $a(v)$, and then scatters value updates along adjacent edges.

There are four UDFs to specify: (1) *gather*(u, v): invoked on each edge adjacent to v to compute a value towards v; (2) *sum*(*combined*, *value*), which accumulates the value along a local edge to a locally combined value; (3) *apply*(D_v, *combined*), which uses the globally combined value and the old value of D_v to compute a new value for D_v; and (4) *scatter*(v, u), which is invoked on each edge adjacent to v (e.g., to activate the neighbor u). As an illustration, consider PageRank computation: (a) *gather*(u, v) simply returns $a(u)/d_{out}(u)$ where both $a(u)$ and $d_{out}(u)$ are accessed from D_u; (b) *sum*(.) simply sums the values returned by *gather*(.); (c) *apply*(.) adjusts the summation by a damping factor (e.g., 0.85) to compute a new value for $a(v)$, and the difference $\Delta a(v)$ between the old and new values; (d) *scatter*(v, u) activates out-neighbor u only if $\Delta a(v) > \varepsilon$.

When a vertex v performs computation, it needs to gather values from every neighbor, but it is possible that only one neighbor u has updated its data D_u (e.g., in later stages of Hash-Min when most vertex values are converged). PowerGraph uses *delta caching* to avoid redundant gather operations: it caches the globally combined values from the previous gather phase for each vertex. The UDF *scatter*(u, v) can optionally return $\Delta a(v)$ to be added to the cached value for v, so that v can bypass the gather phase and call *apply*(.) with the cached value directly.

Edge Partitioning PowerGraph partitions the edges among the machines. Since the edges of a vertex v may be distributed to different machines, D_v (or simply, v) needs to be replicated in multiple machines, which incurs synchronization overhead. Therefore, besides load balancing, the partitioning also attempts to minimize the total number of vertex replicas, which we formalize next. Let $A(v)$ be the set of machines that contain at least one edge adjacent to v; then every machine in $A(v)$ maintains a replica of D_v. We aim to minimize $N = \sum_{v \in V} |A(v)|$, or equivalently, the *replication factor* defined as $N/|V|$. We say that the aim is to minimize the size of *vertex-cut*, in contrast to the more common vertex-partitioning approach that minimizes the size of *edge-cut* (or simply cut).

PowerGraph uses greedy edge placement rules to assign edges to different machines. Let us assume that the edges are processed one after another. When assigning the i-th edge (u, v), for any vertex w, we assume that $A(w)$ is obtained according to the assignment of the previous $(i - 1)$ edges. Then, (u, v) are greedily assigned according to the following three cases:

- If $A(u) \cap A(v) \neq \emptyset$, (u, v) is assigned to the least loaded machine in $A(u) \cap A(v)$;
- If $A(u) \cup A(v) \neq \emptyset$ and $A(u) \cap A(v) = \emptyset$, (u, v) is assigned to the least loaded machine in $A(u) \cup A(v)$;
- If $A(u), A(v) = \emptyset$, (u, v) is assigned to the least loaded machine.

The actual partitioning is performed in parallel, and each machine periodically coordinates with other machines to keep $A(v)$ relatively up to date. Since edges that share end vertices tend to be clustered, the greedy rules tend to reduce the replication factor (i.e., ghost size).

4.2.3 *Maiter: An Accurate Asynchronous Model*

Maiter [15] proposed a computation model called *delta-based accumulative iterative computation* (DAIC), which iteratively updates the vertex values by accumulating the value changes between iterations. Since updates are computed from value changes rather than vertex values themselves, DAIC allows vertex-centric computation to be prioritized by value changes.

Maiter does not adopt a shared memory abstraction as the other systems we introduce in this chapter, but instead using vertex-wise message passing like in Pregel. However, it only supports Pregel algorithms where message combiner is applicable, and the execution can be prioritized to benefit asymmetric value convergence which is more like GraphLab than Pregel.

We introduce Maiter here because its model appears to surpass GraphLab's in several aspects: (1) results obtained by GraphLab are approximate, but Maiter guarantees result exactness; (2) GraphLab's asynchronous model incurs much overhead due to remote data locking/unlocking, which is avoided in Maiter; (3) GraphLab's data organization incurs high memory consumption due to the maintenance of vertex replicas, which is avoided in Maiter as message passing is used.

Maiter is not as popular as GraphLab, possibly because its system is built on the slow MapReduce framework, and the implementation is not released to the public. However, it is a good model to study from the academic perspective, and could serve as a good starting point for developing future systems.

The DAIC Model We now present the DAIC model and illustrate it using PageRank computation as a running example. DAIC requires a vertex-centric algorithm to be formulated into the following two-step update function:

$$\begin{cases} a(v)^{(i)} = & a(v)^{(i-1)} \oplus \Delta a(v)^{(i)} \\ \Delta a(v)^{(i+1)} = & \bigoplus\limits_{u \in \Gamma_{in}(v)} g_{(u,v)}(\Delta a(u)^{(i)}) \end{cases}, \tag{4.1}$$

where $a(v)^{(i)}$ denotes the value of vertex v at the i-th iteration, and $\oplus$ is a generalized summation operator that is commutative and associative. The first equation states that $\Delta a(v)^{(i)}$ is the value change from $a(v)^{(i-1)}$ to $a(v)^{(i)}$, and the second equation states that this change $\Delta a(v)^{(i)}$ can be computed from the value changes of v's in-neighbors in the $(i-1)$-th iteration.

To write a DAIC algorithm, the update function should satisfy two conditions. The first condition is that, the value update function can be formulated into the following form:

$$a(v)^{(i+1)} = \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(a(u)^{(i)} \right) \right) \oplus c(v), \tag{4.2}$$

where $c(v)$ is a constant associated with v. Note that Eq. (4.2) operates on vertex values rather than value changes, and it also defines the formula of $g_{(u,v)}(.)$ as required by Eq. (4.1).

For example, the UDF $v.compute(.)$ in PageRank computation has the following form:

$$a(v)^{(i+1)} = 0.85 \cdot \left(\sum_{u \in \Gamma_{in}(v)} \frac{a(u)^{(i)}}{d_{out}(u)} \right) + 0.15,$$

and therefore, it can be formulated into the form of Eq. (4.2) if we specify $\oplus$ as the summation operator, and specify

$$\begin{cases} g_{(u,v)}(x) = 0.85 \cdot \frac{x}{d_{out}(u)} \\ c(v) = 0.15 \end{cases}, \tag{4.3}$$

The second condition is that, $g_{(u,v)}(x)$ should have the distributive property over $\oplus$:

$$g_{(u,v)}(x \oplus y) = g_{(u,v)}(x) \oplus g_{(u,v)}(y), \tag{4.4}$$

which is obviously satisfied by $g_{(u,v)}(x)$ of Eq. (4.3).

We now show that Eqs. (4.2) and (4.4) guarantee the correctness of applying Eq. (4.1) for computation. By replacing $a(u)^{(i)}$ in Eq. (4.2) with $(a(u)^{(i-1)} \oplus \Delta a(u)^{(i)})$, and using Eq. (4.4), we obtain

$$\begin{aligned} a(v)^{(i+1)} &= \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(a(u)^{(i-1)} \oplus \Delta a(u)^{(i)} \right) \right) \oplus c(v) \\ &= \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(\Delta a(u)^{(i)} \right) \right) \oplus \\ &\quad \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(a(u)^{(i-1)} \right) \right) \oplus c(v), \end{aligned}$$

and using Eq. (4.2), we obtain

$$a(v)^{(i+1)} = \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(\Delta a(u)^{(i)} \right) \right) \oplus a(v)^{(i)},$$

or equivalently,

$$\Delta a(v)^{(i+1)} = \bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)}\left(\Delta a(u)^{(i)}\right),$$

which is consistent with Eq. (4.1).

Finally, $a(v)^{(0)}$ and $\Delta a(v)^{(1)}$ should be initialized so that

$$a(v)^{(0)} \oplus \Delta a(v)^{(1)} = a(v)^{(1)} = \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)}\left(a(u)^{(0)}\right)\right) \oplus c(v).$$

For example, in PageRank computation, we may set $a(v)^{(0)} = 0$ and $\Delta a(v)^{(1)} = 0.15$.

The DAIC model is expressive enough to represent many other Pregel algorithms. For example, Hash-Min can be formulated for DAIC by specifying $\oplus$ as taking the minimum, $g_{(u,v)}(x) = x$, $a(v)^{(0)} = \infty$ and $\Delta a(v)^{(1)} = v$.

Asynchronous Execution Zhang et al. [15] proved that Eq. (4.1) is equivalent to the following asynchronous operations. Specifically, each vertex v maintains two fields $a(v)$ and $\Delta a(v)$. Whenever a delta message $m = g_{(u,v)}(\Delta a(u))$ sent from v's in-neighbor u is received by v, v sets $\Delta a(v) \leftarrow \Delta a(v) \oplus m$. When v performs computation, it (1) sets $a(v) \leftarrow a(v) \oplus \Delta a(v)$, (2) sends $g_{(v,w)}(\Delta a(v))$ to each out-neighbor $w \in \Gamma_{out}(v)$ (if it is not 0), and (3) clears $\Delta a(v)$ back to 0. Note that 0 here refers to the identity element of the $\oplus$ operator (i.e., $x \oplus 0 = x$). The only race-condition is that the update to $\Delta a(v)$ should be atomic.

As an illustration using our PageRank example, whenever a vertex v receives a delta message m, it will add it to $\Delta a(v)$. When v performs computation, it adjusts the received cumulative delta value $\Delta a(v)$, by multiplying it with $0.85/d_{out}(v)$, which is then broadcast to every out-neighbor u (to be added to $\Delta a(u)$); then, v clears the processed cumulative delta value by setting $\Delta a(v)$ to 0, for accumulating more delta values.

Maiter supports prioritized execution. For example, in PageRank computation, each machine may choose the top-1% vertices with the highest $\Delta a(v)$ for vertex-centric computation at each time, and repeat this operation until the terminate condition holds. The master periodically broadcasts a progress request signal to all machines asking for the convergence progress of each machine, and makes a global termination decision based on the responses.

4.3 Single-PC Disk-Based Systems

While Pregel-like message-passing systems are the most popular among distributed vertex-centric systems, the shared memory abstraction is dominant among single-PC big graph systems. After all, the overheads in GraphLab for simulating a shared memory abstraction, such as ghost state synchronization and remote locking, no longer exist in a single PC. This section reviews four popular single-PC systems that expose shared memory abstraction, but perform efficient out-of-core execution to limit memory usage. Their four execution models reflect the four most popular ways to stream a disk-resident big graph. There are also variants specially designed for new hardware technology such as SSD [4, 16] and GPU [6, 17], which are out of the scope of this book and their discussions can be found in [13].

4.3.1 GraphChi

GraphChi [7] was proposed as a single-PC counterpart to distributed GraphLab, and keeps the GAS programming model. GraphChi is PC-friendly since it loads a disk-resident graph part by part into the memory for processing, and is efficient for moderate-sized graphs due to the elimination of network communication.

Edge-Scope GAS Programming GraphChi adopts a simplified version of the GAS programming model, where a vertex v only has access to its *edge-scope* during computation, including (1) $D_{(u,v)}$ for all $u \in \Gamma_{in}(v)$, (2) D_v, and (3) $D_{(v,w)}$ for all $w \in \Gamma_{out}(v)$, but the model is sufficient to cover a broad range of vertex-centric graph algorithms. For example, in PageRank computation, a vertex v may distribute $a(v)/d_{out}(v)$ to every out-edge (i.e., $D_{(u,v)} \leftarrow a(v)/d_{out}(v)$); and v may update $a(v)$ by summing the values from all in-edges, and adjusting it by a damping factor (e.g., 0.85). To realize this programming model, it is important to make sure that when v performs vertex-centric computation, the data values of v and all its in-edges and out-edges are in main memory. Meanwhile, although $D_{(u,v)}$ will be written by u and read by v, we only want to keep one copy of $D_{(u,v)}$ on secondary storage to avoid value synchronization. We will see how GraphChi achieves these requirements next.

Shard-Based Processing GraphChi requires that the IDs of the vertices in a graph with $|V|$ vertices should be numbered as $1, 2, \ldots, |V|$. The IDs are partitioned into P disjoint intervals, $I_1, \ldots, I_P$. All vertices whose IDs fall into an interval I_i constitute a shard (also denoted by I_i for simplicity). We remark that this ID-interval based vertex partitioning is inherited by all subsequent single-PC systems, to be presented in the next few subsections.

In GraphChi, each shard I_i stores not only (1) D_v of every $v \in I_i$, but also (2) all in-edges to vertices in I_i, i.e., (u, v) (containing $D_{(u,v)}$) for all $v \in I_i$ and $u \in \Gamma_{in}(v)$. The in-edges in a shard are stored in the order of their source u's vertex ID.

At each time, all vertices in a shard are loaded from secondary storage into memory for batch processing. To load a shard I_i for processing, for any vertex $v \in I_i$, the data of v and its in-edges are already in memory, but we still need to load the data of every out-edge (v, w) into memory. For this goal, GraphChi loads out-edges of vertices in I_i from every other shard I_j ($j \neq i$) on secondary storage. Since in-edges (v, w) of all vertices $w \in I_j$ are already ordered by source v, all out-edges of vertices $v \in I_i$ are stored consecutively in I_j and can be loaded with only one sequential read. The updated edge values are then written back to the shard I_j by one sequential write. Therefore, processing a shard I_i takes one sequential loading of I_i and $(P - 1)$ sequential reads and writes of out-edges. GraphChi processes every shard once in each iteration, and thus requires $\Theta(P^2)$ non-sequential seeks to secondary storage.

If D_v of all vertices $v \in V$ can fit in memory, GraphChi supports a more efficient semi-streaming model, where a vertex v updates D_v by directly accessing D_u of every in-neighbor $u \in \Gamma_{in}(v)$, eliminating the need of transmitting values through adjacent edges. This optimization saves a lot of disk-IO cost, since otherwise $D_{(u,v)}$ of an edge (u, v) needs to be written by u's shard and read by v's shard. In fact, this vertex-scope abstraction is extended to handle the case where D_v of all vertices $v \in V$ cannot fit in memory by VENUS [1], which we will introduce in Sect. 4.3.3.

Shard Preparation One drawback of GraphChi is the requirement of expensive preprocessing to prepare shards. Specifically, a user needs to first preprocess a graph to make sure that vertex IDs are numbered as 1, 2, ..., $|V|$. Then, GraphChi takes one pass over the disk-resident graph to collect the in-degree of every vertex. Using the in-degree information, GraphChi then divides the vertices into P intervals with approximately the same number of in-edges, to form P shard files. In-edges in each shard file are sorted by source vertex. Finally, the in-degree and out-degree of every vertex are written to a binary degree file for later use.

So far, the preprocessing step guarantees that each shard file has roughly the same size. However, when loading a shard I_i into memory for processing, we still need to load out-edges of vertices in I_i from other shards, and if many vertices in I_i have high out-degree, the number of out-edges to load may exceed the memory space. To solve this problem, GraphChi further divides an interval into sub-intervals, using the information loaded from the degree file, so that each sub-interval can be processed in memory. The use of sub-intervals allows the same set of shard files to be reused by machines of different memory sizes.

Other Features GraphChi supports graph mutation. Specifically, each shard I_i has an edge-buffer B_i for appending new in-edges to vertices in I_i, while removed edges are simply flagged and ignored. After an iteration, each shard I_i is merged with its edge buffer B_i to form a new shard, and if the shard becomes too large, it will be split into two shards.

GraphChi also supports selective scheduling so that vertices can be processed with different frequencies. Specifically, in an iteration, a vertex that performs computation can flag a neighboring vertex to be updated in the next iteration. A bitmap is used to record whether each vertex is flagged in each iteration, and GraphChi skips the computation on unflagged vertices. The effectiveness of selective scheduling for sparse computation is limited, since a whole shard needs to be loaded even if only one vertex in it is flagged.

4.3.2 X-Stream

The second popular single-machine system is X-Stream [12], which follows a different design from GraphChi and does not require preprocessing. X-Stream can also run in memory, or on SSD, both of which exhibit very good performance [12]. However, when hard disk is used, X-Stream is usually slower than GraphChi [1, 33], possibly due to the lack of a mechanism to skip streaming edges of inactive vertices.

Computation Model Like GraphChi, X-Stream adopts an edge-scope GAS programming model, but the execution model is different. X-Stream eliminates the need for sorting edges, but instead streams a completely unordered list of edges. Streaming disk-resident (resp., memory-resident) edge lists allows X-Stream to fully utilize the high sequential bandwidth of a hard-disk (resp., main memory) with the help of a main-memory buffer (resp., CPU cache).

The computation model of X-Stream is edge-centric, and each iteration consists of two sequential passes. We now illustrate the two passes by considering PageRank computation (with damping factor of 0.85), assuming that the vertex states are all maintained in an in-memory array, and edges are stored on disk in random order.

The first pass streams the edge list and performs edge-centric scattering. For each edge (u, v), X-Stream gets $a(u)$ and $d_{out}(u)$ from the in-memory state of u, and computes an **update** value $a(u)/d_{out}(u)$ for the edge (u, v), which is then appended to an update stream on disk. After the first pass, X-Stream re-initializes all in-memory vertex values $a(v)$ as $(1 - 0.85)/|V|$.

The second pass streams the list of updates and performs edge-centric gathering. For each update $m = a(u)/d_{out}(u)$ of an edge (u, v), we add $0.85 \cdot m$ to the in-memory value field $a(v)$. After the second pass, the PageRank values of all vertices are advanced for one iteration.

To sum up, there are three important UDFs in X-Stream: (1) *init*(.), which indicates how to re-initialize the value of a vertex before the gathering pass; (2) *apply_one_update*(.), which indicates how to update the value of a vertex using a gathered update; and (3) *generate_update*(.), which indicates how to generate the update value of an edge (u, v) using u's state (and possibly the edge value of (u, v)).

Out-of-Core Execution The above example assumes that all vertex states fit in main memory. When this assumption does not hold, vertices are partitioned into P disjoint intervals like in GraphChi, so that all vertices in an interval I_i fit in memory and constitute a vertex partition, which we denote by V_i. Since a vertex partition does not keep edges, given the same memory space constraint, a partition in X-Stream contains more vertices than a shard in GraphChi. Therefore, the number of partitions in X-Stream is much smaller than the number of shards in GraphChi.

Each vertex partition V_i is also associated with an edge partition E_i, which contains all out-edges of vertices in V_i, i.e. $\{(u, v) \in E \mid u \in V_i\}$. In the first pass for edge-centric scattering, each vertex partition V_i is loaded into memory to generate updates by streaming the edge partition file E_i. This pass writes to P update files, $U_1, \ldots, U_P$, one for each partition. When an update on edge (u, v) (where $v \in V_j$) is generated, the update is appended to the update file U_j. In the second pass for edge-centric gathering, each vertex partition V_i is loaded into memory to update vertex values by streaming the update file U_i generated by the first pass.

The edge partition can be generated in one pass over the whole edge list, and by appending each edge (u, v) where $u \in V_i$ to the edge file E_i. This is the only preprocessing required by X-Stream.

X-stream also supports in-memory execution to fully exploit the parallelism of all available cores by considering cache locality, which we omit here.

Weakness As admitted in [12], X-Stream is inefficient for *graphs whose structure requires a large number of iterations*, such as a large graph diameter. This is because each iteration streams all edges of a graph even if only a small number of vertices participate in computation. This weakness is also observed by other works such as [14].

Scaling-Out To utilize the aggregate disk bandwidth of multiple machines, Chaos [11] scales out X-Stream by distributing the partitions (i.e., (V_i, E_i) pairs) evenly to multiple machines for processing, so that each machine only streams part of the partitions. In Chaos, a master keeps track of the vertices and edges of every partition, and the generated updates towards every partition; while a computing thread sends requests to the master for the necessary data for processing a partition. This approach assumes that *cluster network bandwidth far outstrips storage bandwidth*. In fact, Roy et al. [11] reported that Chaos only achieves good performance by using large-SSD machines connected by 40 Gigabit Ethernet, and the performance is undesirable when Gigabit Ethernet is used.

Chaos also supports work stealing for load balancing. When a machine M finishes all its assigned partitions in an iteration, it may send request to another machine M' to steal the workload of processing M''s partition $P_i = (V_i, E_i)$. If M' accepts the request, M may pull the data of P_i for processing. Since both M and M' may be processing P_i, the master should assign disjoint sets of edges (during scatter) and updates to M and M'. Multiple accumulated values of a vertex (processed by different machines) need to be combined before completing the gather phase.

4.3.3 VENUS

The next two systems we shall introduce are not as popular as GraphChi and X-Stream, likely because open-source systems are not provided. However, they propose different computation models from those of GraphChi and X-Stream, and are of value to system researchers and practitioners when developing their own systems.

This subsection introduces VENUS [1], which exposes a vertex-scope GAS programming model to users, and is restricted to computation in a static graph (i.e., graph mutation is not supported).

Programming Model VENUS adopts a vertex-scope GAS programming model where a vertex v only accesses D_v, and D_u for all $u \in \Gamma_{in}(v)$. For example, in PageRank computation, when a vertex v computes D_v by summing $a(u)/d_{out}(u)$ of every $u \in \Gamma_{in}(v)$, it directly accesses the vertex value of u. This is in contrast to the approach of GraphChi and X-Stream, where u first distributes the value (or update) to edge (u, v), and then v gets the value (or update) from (u, v). As a result, this approach saves the disk-IO cost of writing $O(|E|)$ values (or updates) to the edges.

Data Organization In the above programming model, each iteration only needs to scan the static graph topology data once, but the value of a vertex may be accessed multiple times to update the value of its out-neighbors. Therefore, VENUS separates the read-only structure data from mutable vertex values on disk: the structure data are streamed during the computation which only requires a small in-memory buffer, and thus the available main memory can be used to cache as many mutable vertex values as possible (for efficient access during the computation). This is in contrast to GraphChi where all adjacent edges of a shard need to be loaded into memory before the shard can be processed. This new computation model is called *vertex-centric streamlined processing* (VSP).

Like in GraphChi, vertices are partitioned into P disjoint intervals according to their IDs, $I_1, \ldots, I_P$. Each interval I_i defines a g-shard and a v-shard as follows: (1) the g-shard stores all the edges (and the associated read-only attributes) with destinations in I_i, and (2) the v-shard contains all vertices in the corresponding g-shard, including the source and destination of each edge. Moreover, edges in a g-shard are ordered by destination, and thus the in-edges of each vertex are stored consecutively. In other words, the g-shard of I_i stores $\Gamma_{in}(v)$ (and the associated edge attributes) of every vertex $v \in I_i$ one by one. All g-shards are further concatenated to form the *structure table*, which is streamed during VSP.

Computation Model In each iteration, the VSP model processes vertices of $I_1, \ldots, I_P$ one by one, where the in-edges of each vertex is streamed from the structure table. If the values of all vertices fit in memory, VENUS supports a semi-streaming in-memory mode similar to that of GraphChi. Otherwise, VENUS supports two IO-friendly algorithms described as follows.

The first algorithm materializes all v-shard values as a view for each shard. To process the vertices in I_i, the relevant vertex values are loaded from the v-shard of I_i. After all vertices in I_i are processed, their new values need to be updated to all the relevant v-shards. This is because, a vertex $u \in I_i$ may be the in-neighbor of another vertex $v \in I_j$ ($j \neq i$), and thus u's value is included in the v-shard of I_j. VENUS orders the values of the vertices in each v-shard by their vertex ID, and thus for each v-shard I_j, the values of those vertices whose IDs are in interval I_i are stored consecutively and can be updated by one sequential write.

To eliminate the cost of materializing all (possibly replicated) vertex values in v-shards, the second algorithm merge-joins each v-shard with the table of all vertex values. In this case, the v-shard of I_i only stores the IDs of the relevant vertices, and their values are obtained by joining the v-shard with the vertex value table by vertex ID. Since data in both a shard and the vertex value table are ordered by vertex ID, merge-join is applicable and efficient. The join results include all relevant vertex values and are cached in memory for processing I_i.

4.3.4 GridGraph

GridGraph [18] partitions the adjacency matrix of a graph by a grid. Similar to GraphChi, vertices are partitioned into P intervals. Using the P intervals, the vertex (value) vector is broken into P chunks, and the adjacency matrix (denoted by A) is broken into a grid of $P \times P$ edge blocks. Here, a block (I_a, I_b) includes all edges with source in interval I_a and destination in interval I_b, and are stored (in arbitrary order) as a file on the disk. Note that unlike GraphChi, X-Stream and VENUS, the edges of a vertex may be stored in multiple files in GridGraph.

Computation Model GridGraph combines the scattering and gathering phases into one "streaming-apply" phase, which streams every edge and applies the generated update instantly onto the vertex. Let $a_i(v)$ be the vertex value of v at iteration i. In an iteration i, GridGraph processes each block (I_a, I_b) once if any vertex $v \in I_a$ is active. To process the edges in block (I_a, I_b), GridGraph pins the following data in main memory: (1) the vertex value chunk of I_a for iteration i, (2) the vertex value chunk of I_b for iteration $(i + 1)$. The update on an edge (u, v) is computed from $a_i(u)$ (obtained from the chunk of I_a) and the value of edge (u, v), which is then aggregated to $a_{i+1}(v)$ in the chunk of I_b.

GridGraph processes the blocks in the grid column by column, and for each column, the blocks are processed from top to bottom. The benefit of column-oriented computation is that, all blocks in a column for interval I_b updates the vertex values in chunk I_b, and thus chunk I_b can be pinned in memory during the processing of the whole column, and flushed to the disk after the column is processed. Therefore, the processing of each column needs to read P chunks of I_a (i.e., $O(|V|)$ vertex values) and write one chunk of I_b. Accordingly, each iteration needs to read $O(P \cdot |V|)$ data of vertex chunks and write $O(|V|)$ data of vertex chunks, in addition to streamingly reading $O(|E|)$ edges in the grid. The efficiency of this computation model lies in the fact that the data written to disk is linear to the vertex number, rather than the edge number as in GraphChi and X-Stream.

Other Optimizations GridGraph can also run in asynchronous mode, in which case for a block (I_a, I_b), there is no concept of iteration number for the corresponding vertex chunks I_a and I_b, and they are just the states of subsets (i.e. intervals) of vertices that can be both read and updated. This allows faster convergence for algorithms like Hash-Min.

Recall that the edges in a block are streamed during the "streaming-apply" computation, which only requires a small in-memory buffer to achieve full sequential bandwidth for large block files. However, some blocks can be very sparse, and the small file size may lead to frequent disk seeks. Therefore, the blocks are appended into one large file for streaming, with block boundaries recorded so that GridGraph knows when to pin/unpin a vertex chunk.

To utilize CPU cache locality, each block (I_a, I_b) in main memory can be further partitioned by a $Q \times Q$ grid, and accordingly, each vertex chunk (i.e., I_a and I_b) is partitioned into Q smaller sub-chunks such that each sub-chunk fits into the last-level CPU cache. This in-memory grid of (I_a, I_b) is also processed in a column-oriented manner to reduce cache misses.

References

1. J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. VENUS: vertex-centric streamlined graph computation on a single PC. In *ICDE*, pages 1131–1142, 2015.
2. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
3. M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.
4. W. Han, S. Lee, K. Park, J. Lee, M. Kim, J. Kim, and H. Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.
5. G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.
6. F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *HPDC*, pages 239–252, 2014.
7. A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.
8. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.
9. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
10. Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.
11. A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.
12. A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.
13. D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1–2):1–195, 2017.
14. D. Yan, Y. Huang, J. Cheng, and H. Wu. Efficient processing of very large graphs in a small cluster. *CoRR*, abs/1601.05590, 2016.
15. Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.*, 25(8):2091–2100, 2014.
16. D. Zheng, D. Mhembere, R. C. Burns, J. T. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.
17. J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, 2014.
18. X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.

Part II
Think Like a Graph

Chapter 5
Block-Centric Computation

5.1 Comparison: Block-Centric vs. Vertex-Centric

Motivations for Block-Centric Computation The vertex-centric model requires one superstep to propagate data for merely one hop, and is thus mainly designed to process small diameter graphs like social networks. However, many real big graphs have a large diameter, such as continental road networks and terrain meshes. Even non-spatial graphs may have a large diameter, such as web graphs which exhibit spatial locality: a local web page is more likely to link to another local web page than a web page elsewhere (e.g., abroad). For example, Salihoglu and Widom [2] reported that it takes 4546 and 6509 supersteps to compute the strongly connected components of two web graphs *uk-2005* and *sk-2005*. To solve this problem, they designed an algorithmic optimization called *Finishing Computations Serially* (FCS) which monitors the number of active vertices, so that once this number is small enough, these active vertices (and their adjacency lists) are sent to the master, and serial computation is performed on the constructed subgraph. After applying FCS, the number of supersteps is reduced to 3278 and 2857, respectively, which is still very large.

A satisfactory solution to this problem is to extend the vertex-centric computation model with a novel block-centric computation model, the idea of which is briefly introduced as follows. Specifically, the vertices of a graph are partitioned into multiple blocks, such that each block has a strong cohesion: a vertex in a block B is more likely to connect to another vertex in B than to a vertex in another block. All vertices in a block is assigned to one worker. When vertices in a block B receive incoming messages, they do not just update their own states using these messages, but also propagate the state updates through all vertices in B until convergence. Since in-block state propagation is performed in serial without communication, the additional computation overhead incurred is negligible compared with the significant reduction in message number and in superstep number.

D. Yan et al., *Systems for Big Graph Analytics*, SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-58217-7_5

Block-centric computation well solves the problem of large graph diameter: for example, Yan et al. [5] reported that single-source shortest path computation on the USA road network takes 10,789 supersteps (and 2832 s) in the vertex-centric model, and finishes in only 59 supersteps (and 11 s) with block-centric computation.

Challenges Implementing the block-centric model faces two major challenges. (1) An input graph needs to be pre-partitioned into blocks, but graph partitioning is expensive, especially for big graphs. (2) In Pregel, the worker that a vertex resides in can be computed directly from its vertex ID, but when block-centric computation is used, it is non-trivial to find a function that maps the IDs of all vertices in a block B to the ID of the worker that contains B. There also exist other challenges, such as how to define the stop (or convergence) condition of the computation.

Existing Solutions Giraph++ [3] pioneered the idea of block-centric computation, which is termed "graph-centric" or "think like a graph". The term "graph" here is equivalent to the concept of "block" we previously described. The input graph is partitioned into blocks by a METIS-like algorithm [1], and vertex IDs are recoded by an independent MapReduce job, so that the worker that a vertex resides in can be directly computed from the new vertex ID.

Blogel [5] further allows each block to contain data structures like an adjacency list and a value, so that computation can be directly performed in the unit of blocks without the involvement of individual vertices. Since there are much less blocks than vertices, the workload is significantly reduced. In Blogel, each block is a connected subgraph, and a worker contains multiple blocks. This overpartitioning approach allows vertices and blocks to be distributed among workers in a balanced manner. The ID of each vertex v is expanded to also store the IDs of the block and the worker that contain v, so that it is trivial to determine whether two vertices are in the same block, and which worker a vertex resides in. Blogel also proposed partitioning algorithms that are way more efficient than METIS-like methods.

The block-centric model has also been applied in single-machine in-memory graph processing. For example, GRACE [4] partitions vertices into blocks by METIS, so that each block fits in the CPU cache. All vertices in a block are processed together (possibly until convergence) without cache miss, before processing another block. This block-centric solution improves cache locality and mitigates the problem of limited memory bandwidth. Unlike Giraph++ and Blogel, GRACE only requires a user to specify the vertex-centric computation logic, and the block-centric computation is treated as a proper scheduling of vertex-centric computation inside each block.

Relation to Delta-Based Accumulative Iterative Computations (DAIC) Recall Maiter's DAIC model from Sect. 4.2.3. The DAIC model actually can be implemented in the block-centric model for efficient computation. For example, consider PageRank computation, and assume that each block b can call a UDF *compute*(.) where it can send messages to vertices. Then, *b.compute*(.) may repeatedly iterate over its vertices as follows until their vertex values do not change much.

In *b.compute*(.), when the value of a vertex v in block b is updated, for every out-neighbor $w \in \Gamma_{out}(v)$, (1) if w is not in b, a delta message $g_{(v,w)}(\Delta a(v))$ is sent to w (which can actually be combined to w's local mirror that gets committed when *b.compute*(.) finishes); (2) otherwise, $g_{(v,w)}(\Delta a(v))$ is directly added to $\Delta a(w)$, which will be processed later when *b.compute*(.) continues to process u. In fact, this algorithm has been implemented in Giraph++ [3].

In the rest of this chapter, we introduce Blogel, the state-of-the-art block-centric framework; we then give a tutorial on how to get started using Blogel.

5.2 The Blogel System

We now introduce the Blogel[1] system, a member of the BigGraph@CUHK toolkit.

Programming Interface In addition to the *Vertex* base class of a Pregel-like system, Blogel has another base class, *Block*, which takes the user-defined vertex subclass as a template argument that indicates the data type of vertices in a block. Like a vertex object, a block object b has a flag $active(b)$ and can vote to halt; b can also maintain its own attributes, such as a value $a(b)$ and a block-level adjacency list $\Gamma(b)$ that links to other blocks. In addition, b can access an array of its own vertices, which is actually a subarray of the vertex array maintained by the worker that b resides in.

Both a vertex object and a block object can send two types of messages: those towards another vertex, and those towards another block. Accordingly, two message combiners can be specified for combining the respective types of messages. The *Block* class also has a UDF *compute*(.), to be called by each block during the computation, whose input are those messages towards the current block.

After a worker loads its vertices from HDFS, it groups them by their block ID to construct an array of block objects automatically (see Fig. 5.1 for an illustration). Before computation begins, each block b calls another UDF *block_init*(), which specifies how to initialize the attribute of b (e.g., from its vertices).

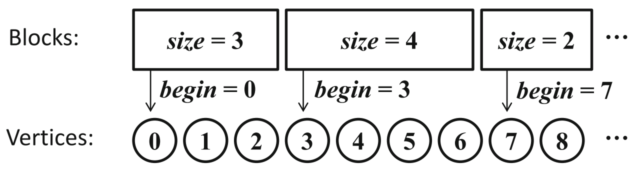


Fig. 5.1 The block arrays and vertex array in a worker

[1] http://www.cse.cuhk.edu.hk/blogel/.

Execution Modes Three execution modes are supported in Blogel: (1) **V-mode**, which is exactly like the vertex-centric model, but is usually more efficient since vertices are grouped into blocks with strong cohesion, and messages transmitted between two vertices in the same block do not incur network communication; (2) **B-mode**, where only blocks call *compute*(.) functions and message passing only happens among blocks, and computation terminates when all blocks are inactive and there are no pending messages; (3) **VB-mode**, where in a superstep, all active vertices call *Vertex::compute*(.) first, and then all active blocks call *Block::compute*(.), and computation terminates when all vertices and blocks are inactive and there are no pending messages.

We illustrate how to write a B-mode algorithm by considering Hash-Min. Instead of letting vertices broadcast the smallest vertex ID they have seen, we let blocks broadcast the smallest block ID they have seen. In *b.block_init*(), we construct $\Gamma(b)$ from $\Gamma(v)$ of every vertex in b as follows: if $u \in \Gamma(v)$, then we add the block of u (available in u's ID) to $\Gamma(b)$. When reporting results, each vertex sets its value $a(v)$ as the value of its block b, i.e., $a(b)$. Since vertices in a block are all connected, all vertices with the same value constitute a connected component. This B-mode algorithm is more efficient than the vertex-centric Hash-Min algorithm since there are much less blocks than vertices.

While B-mode algorithms are not supported by Giraph++, VB-mode algorithms have a similar flavor to *GraphPartition.compute*(.) in Giraph++, except that a VB-mode algorithm can separate the computation logic related to blocks and vertices to *Block::compute*(.) and *Vertex::compute*(.), respectively.

As an illustration of VB-mode, consider the computation of single-source shortest paths from a source vertex s, where we denote the weight of each edge (u, v) by $w(u, v)$. In this algorithm, the vertex value $a(v)$ keeps an estimated distance from s to v, and messages are only sent to vertices (not blocks). In a superstep, (1) *Vertex::compute*(.) is first called by vertices that receive messages, where each vertex v receives the distance estimations from its neighbors in other blocks. If the smallest message is less than $a(v)$, v updates $a(v)$ as the new value and remains active. Otherwise, v votes to halt. Then, (2) *Block::compute*(.) is called by blocks that contain active vertices. This does not need user intervention since Blogel also activates the block of a vertex v when v is activated. In *b.compute*(.), b first collects all its active vertices into a priority queue Q (since their values were updated by *Vertex::compute*(.)), and votes them to halt (thus all vertices are halted at the end of a superstep). Then, b runs Dijkstra's algorithm, where a vertex v with the smallest value $a(v)$ is popped from Q each time, and for each neighbor $u \in \Gamma_{out}(v)$, (a) if u is in b, $a(u)$ is updated with $(a(v) + w(v, u))$ and u is added to Q (or u's position in Q gets adjusted), while (b) if u is not in b, a message $(a(v) + w(v, u))$ is sent to u.

Graph Partitioning Blogel supports three kinds of partitioners for processing a graph data, whose output can be used as the input data of a block-centric program. The first one is URL partitioner, which is only applicable to web graphs where each vertex contains a URL. The URL partitioner groups vertices under the same host name (or domain name) into one block. The second one is 2D partitioner, which

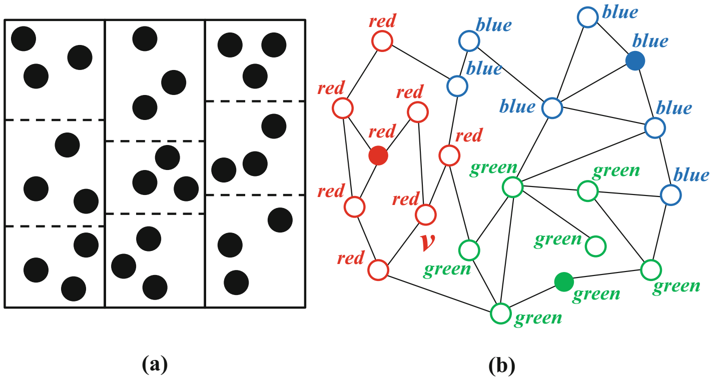


Fig. 5.2 Blogel partitioners. (**a**) 2D partitioner, (**b**) graph Voronoi diagram

is only applicable to spatial graphs where vertices have 2D coordinates. The 2D partitioner first let every worker sample a small fraction of vertices, which are gathered at the master that then computes a balanced spatial partitioning from these samples as Fig. 5.2a illustrates. The spatial partitioning is broadcast back to every worker, which then assigns each of its vertex to a proper partition. The workers then exchange vertices with each other, so that each worker W contains all vertices in those partitions that are assigned to W. Since we require a block to be connected, each worker then runs BFS in each of its partitions, to get connected subgraph blocks.

For a general graph, Blogel provides a *graph Voronoi diagram* (GVD) partitioner for computing blocks. Since graph Voronoi diagram is only defined for undirected graphs, a directed graph will be converted into an undirected one before partitioning. This conversion is simple through vertex-centric computation: in Superstep 1, every vertex v sends its own ID to every out-neighbor $u \in \Gamma_{out}(v)$; in Superstep 2, every vertex u collects all messages to form $\Gamma_{in}(u)$. Finally, the neighbors of v in the converted graph is obtained as $\Gamma(v) = \Gamma_{out}(v) \cap \Gamma_{in}(v)$.

The idea of GVD partitioning is illustrated in Fig. 5.2b. Firstly, a small fraction of vertices are sampled as sources (solid circles in Fig. 5.2b). Then, multi-source breadth-first traversal is performed from the sampled sources in vertex-centric model, which sets the vertex value $a(v)$ as the source closest to v. In $v.compute(.)$ (of the GVD partitioner), if (1) $a(v)$ is already assigned (i.e., a closer source has reached it), v votes to halt directly; (2) otherwise (v is reached by a source for the first time), v sets $a(v)$ as the first received source ID, and broadcasts it before voting to halt. The multi-source traversal is fast since each vertex broadcasts messages at most once, i.e., the total message number is $O(|E|)$. For the example in Fig. 5.2b, the vertices are grouped into three blocks, represented by three different colors.

However, since sources are randomly sampled, some blocks may contain too many vertices. The GVD partitioner then marks the states of those vertices back as "unassigned", and samples sources from the remaining unassigned vertices with an increased sampling probability. Then, multi-source BFS is performed among the unassigned vertices again to obtain new blocks. This process is repeated until some quality requirement is met. Finally, Hash-Min is run on all unassigned vertices, and each connected component is treated as a block. The last step is necessary, since a graph may contain many small connected components, and it is likely that none of the vertices in a component is ever sampled due to the low sampling probabilities.

Experiments in [5] demonstrate that the GVD partitioner scales almost linearly with the graph size, and the partitioning time is comparable to the time for loading the input graph from HDFS and dumping the partitioned graph to HDFS.

5.3 Get Started with Blogel

Blogel can be accessed from http://www.cse.cuhk.edu.hk/blogel/.[2] The process of deploying and running Blogel is almost the same as that for Pregel+ described in Sect. 3.2, and the instructions can also be found at the first two links on Blogel's *Documentation* page.[3]

The system and application code can be downloaded from Blogel's *Download* webpage.[4] The system code contains three libraries (or folders), where like in Sect. 3.3, "utils" implements the basic functionalities used by upper-level systems, and "basic" implements the original computation model of Pregel. There is a new library "blogel", which implements the block-centric framework of Blogel and is the focus of our presentation next.

We remark that while this tutorial is very helpful in getting you started using Blogel, it is also necessary to read the last two links on Blogel's *Documentation* page, one about the system architectural design[5] and the other about Blogel's API.[6]

5.3.1 Blogel Graph Partitioners

To run a graph analytics job in block-centric model, the first thing is to partition an input graph into blocks. We assume that the input graph is stored as a file (or a directory of files) on HDFS, where each line stores the information of a vertex just like the input to Pregel+. Ideally, we would like to have a distributed program that takes this input, partitions the graph, and then outputs the partitioned graph that can directly function as the input to a block-centric program.

[2]You may also search "Blogel" in Google.

[3]http://www.cse.cuhk.edu.hk/blogel/documentation.html.

[4]http://www.cse.cuhk.edu.hk/blogel/download.html.

[5]http://www.cse.cuhk.edu.hk/blogel/overview.html.

[6]http://www.cse.cuhk.edu.hk/blogel/api.html.

Blogel provides three kinds of *partitioners* to partition the vertices of the input graph into blocks, whose output can be directly used for block-centric computation. However, we require that the number of workers used for graph partitioning be the same as that used for block-centric computation. This is because each worker (let it be worker *i*) of a partitioner outputs a file of blocks to HDFS, which will be directly loaded into worker *i* of a block-centric program later.

Graph Voronoi Diagram (GVD) Partitioner This partitioner is defined by the header file "blogel/Voronoi.h". Recall that GVD partitioner first uses multi-source BFS to compute the graph Voronoi diagram for multiple rounds, and then runs a round of Hash-Min at last. These are all vertex-centric computation, where each vertex is an object of the system-defined *BPartVertex* class (Line 49), which is a subclass of *Vertex* (defined in "basic/Vertex.h") with integer ID and value type defined by the type *BPartValue* (Line 17).

To use the GVD partitioner, users need to know the structure of *BPartValue* class, which has four fields (assuming that the current vertex is v):

1. *color*, which is used to keep v's block ID (e.g., the ID of the closest sampled source from v) computed by the GVD partitioner;
2. *neighbors*, which keeps a list of integer IDs for v's neighbors, and should be set by users when parsing a line into a vertex of type *BPartVertex*;
3. *nbsInfo*, which keeps a list of new IDs for v's neighbors computed by the GVD partitioner, where each ID is of type *triplet* (defined in "blogel/BType.h") that consists of three integer fields: vertex ID vid, block ID *bid* and worker ID *wid*;
4. *content*, a string for storing other information irrelevant to GVD partitioning but may be useful in block-centric computing later.

BPartWorker (Line 114) is the worker class that performs vertex-centric GVD partitioning, which provides a function *run*(.) (Line 666) like Pregel+'s worker class for starting the execution of a partitioning job. The *run*(.) function performs the following operations after graph loading (can be safely skipped of you are not interested in the implementation):

1. If the input graph is directed, it is converted into an undirected graph (Lines 710–711).
2. Multiple rounds of multi-source BFS is the run, before which message combiner is set to only keep one message to each target vertex (Line 715).
3. Then, if there are still unassigned vertices, Hash-Min is run over these vertices by calling the *subGHashMin*() function at Line 790, which is defined at Line 346 and changes the message combiner to only keep the smallest message (Line 350).
4. So far, every vertex is assigned a block ID, but the vertices of a block may scatter among multiple workers. The function *greedy_assign*() (Line 797) is then called to gather and aggregate block information from all workers, and to compute a balanced block-to-worker assignment.
5. Then, *nbInfoExchange*() (Line 806) is called to properly set the triplet IDs of the neighbor list *nbsInfo* of each vertex. This is simple since Step 3 (resp. Step 4) already sets the block ID (and worker ID) of each vertex, and each vertex only

needs to broadcast its vertex, block and worker IDs to its neighbors which then collect them into their *nbsInfo* lists.

6. Before graph dumping, *block_sync*() (Line 813) shuffles the vertices into their assigned workers properly.

A user subclasses the *BPartWorker* class to use the GVD partitioner. *BPart-Worker* provides two abstract functions for users to specify:

```
BPartVertex* toVertex(char* line)
void toline(BPartVertex* v, BufferedWriter& writer)
```

The first UDF defines how to parse a line from the input graph data on HDFS into a vertex object of type *BPartVertex* used for GVD partitioning.[7] For each vertex v of type *BPartVertex*, a user only needs to set vertex ID $v.id$, and its adjacency list $v.value().neighbors$ properly, where $v.value()$ is of type *BPartValue*. Additional information such as edge length can be added to $v.value().content$ for later use. After GVD partitioning, for each vertex v, block ID $v.value().color$ and neighbors' triplet IDs $v.value().nbsInfo$ are set. The second UDF specifies how to write these information of v (e.g., as a line) to HDFS, and during this process one may add back information in $v.value().content$ such as edge length.

Before calling *run*(.), a user also needs to set the GVD partitioning parameters properly, though the default parameters work reasonably well. These parameters are defined in "Blogel/BGlobal.h", Lines 10–46. We refer readers to Sect. 7.1 of [5] for the concrete meaning of each parameter and for how to set them properly for your input graph.

Example: Single-Source Shortest Paths Let us look at the GVD partitioner's code for this application.[8] Each line of the input is assumed to be of the following format:

```
vertex_ID x-coordinate y-coordinate \t number_of_neighbors
ne ighbor1_ID edge1_length neighbor2_ID edge2_length...
```

In "blogel_sssp_vorPart.h", we define a subclass of *BPartWorker* called *vorPart* (Line 7). In UDF *toVertex*(.), we append each neighbor ID to the field *neighbors* of the parsed vertex (Lines 25–27), but skip the end length (Line 28). However, the edge lengths are not lost since we keep the whole line to the field *content* (Line 16).[9]

In UDF *toline*(.), we re-parse the line stored in the field *content* (Line 44), and for each neighbor, we find its triplet ID stored the field *nbsInfo* (Lines 38–42 and Line 56), and output it along with the edge length (Line 57). We remark that even though GVD partitioner converts the input from directed into undirected, *content* still contains the list of out-neighbors and so does the line output by *toline*(.). Therefore, the input to block-centric computation is still directed.

[7]That is why a user needs to know the structure of *BPartVertex* and *BPartValue*.

[8]http://www.cse.cuhk.edu.hk/blogel/code/apps/block/sssp/Vor.zip.

[9]We copy the line first, since *strtok*(.) replaces the token splitters with '\0' and thus changes the line. We use *strtok*(.) since C++'s stringstream needs to copy the line into its internal buffer for parsing, which incurs additional overhead.

2D Partitioner This partitioner requires two rounds. The first round partitions the vertices into rectangular cells, while the second round further splits the vertices in each cell into connected blocks.

The first round is defined by header file "blogel/STRPart.h", where the vertex (resp. worker) class is *STRVertex* (resp. *STRWorker*) defined in Line 17 (resp. Line 60). The fields of *STRVertex* are similar to those of *BPartVertex*, but *STRVertex* is not a subclass of *Vertex* (or any class). Therefore, fields like block ID *bid*, neighbor list *neighbors* and *nbsInfo* are *STRVertex*'s direct fields rather than fields of its value field (which is non-existent). Two additional fields are the (x, y)-coordinates (which is obviously necessary for 2D partitioning), which should also be set by users during graph loading.

Similar to *BPartWorker*, a user should subclass *STRWorker* and implement the data import/export UDFs, and then call *run*(.) to start round 1's execution. Remember to set (x, y)-coordinates of a created vertex in the UDF *toVertex(.)* which will be used by the partitioner for spatial partitioning.

A few more words on *STRWorker*'s implementation, which can be safely skipped. In *run*(.), after graph loading, each worker samples a small set of vertices, whose coordinates are gathered at the master to compute a balanced cell partitioning, which is then broadcast back to every worker. This is performed by *getSplits*(.) at Line 590, which is defined at Line 175. Then, each worker sets the cell ID of each vertex in it, using the cell partitioning received. This is performed by *setBlkID*(.) at Line 605, which is defined at Line 263. Finally, *greedy_assign*(), *nbInfoExchange*() and *block_sync*() are called in sequence to distribute the cells evenly among all workers like *BPartWorker* does.

The second round is defined by header file "blogel/STRPartR2.h" and runs in block-centric mode, where each cell is treated as a block. As a result, the program is similar to a regular block-centric program of Blogel, which subclasses the base vertex (resp. block) class *BVertex* (resp. *Block*) to be described shortly.

The vertex (resp. block) class is *STR2Vertex* (resp. *STR2Block*) is defined in Line 29 (resp. Line 49). The value of an *STR2Vertex* vertex v is of type *STR2Value*, which contains the following three fields:

1. *new_bid*: v's block ID to compute, which is the result of BFS over v's cell.
2. *neighbors*: the triplet ID list of v's neighbors. Before partitioning, each triplet ID consists of vertex ID, cell ID and worker ID, which is exactly the output of round 1. After partitioning, the cell ID is replaced by block ID.
3. *content*: a string for storing other information irrelevant to 2D partitioning.

Internally, there is another field *split* (Line 22) that splits *neighbors* into two parts: those within the same cell of v, and those in another cell. This allows intra-cell BFS to only process the small list of in-cell neighbors during computation. The splitting is performed by *STR2Worker*'s function *blockInit*() (Line 173).

Similar to *STRWorker*, a user should subclass *STR2Worker* and implement the data import/export UDFs, and then call *run*(.) to start round 2's execution. The output of *STRWorker* is exactly the input of *STR2Worker*, and the import UDF should parse each line properly to translate the information of each *STRVertex* vertex into an *STR2Vertex* vertex.

A few more words on *STR2Worker*'s implementation, which can be safely skipped. In *run*(.), (1) after graph loading, each cell runs BFS to split itself into connected blocks, and the vertex array is also reordered to be grouped by blocks. This is performed by *superstep1*() at Line 408 which is defined at Line 248. (2) Then, *getPrefix*() at Line 411 (defined at Line 295) let master gather the number of blocks in every worker, and computes for each worker (let it be worker i), the total number of blocks in all workers $j < i$, denoted by *prefix*. (3) Right afterwards, *shiftBlkId*() at Line 412 (defined at Line 315) adds *prefix* to block ID (starting from 0) of every in-cell vertex computed by BFS. This guarantees that block IDs of different cells are disjointly relabeled, and thus the ID of each block is unique. (4) Finally, *superstep2_3*() at Line 413 (defined at Line 326) let each vertex v broadcast its block ID (i.e., field *new_bid*) to its neighbors, so that they can replace v's cell ID in their adjacency lists with the received block ID.

Example: Single-Source Shortest Paths Let us look at the 2D partitioner's code for this application.[10] File "blogel_sssp_STRRnd1.h" (resp. "blogel_sssp_STRRnd 2.h") implements round 1 (resp. round 2) of 2D partitioning, where we define the subclass *STRRnd1* (resp. *STRRnd2*) of the base class *STRWorker* (resp. *STR2Worker*). Note that *STRWorker* requires users to specify a sampling rate, and the example sets it as 1% when running *STRRnd1*. It is strongly recommended to try GVD partitioning and 2D partitioning over the *USA Road Network* dataset on Blogel's *Download* web page, to get an idea of how Blogel's partitioners work.

Partitioning Based on Predefined Block IDs This partitioner is defined by the header file "blogel/BAssign.h", where the vertex (resp. worker) class is *BAssign-Vertex* (resp. *BAssignWorker*) defined at Line 43 (resp. Line 60). Unlike the other partitioners which need to first compute the block ID of each vertex and then shuffle vertices by their blocks, *BAssignWorker* skips the first step and directly ask users to specify the block ID of each vertex in the data import UDF. A *BAssignVertex* vertex has a value of type *BAssignValue* (Line 16), whose field *block* is where users specify the block ID.

Example: PageRank Let us look at the URL partitioner's code for this application.[11] File "blogel_pagerank_urlPart.h" implements this partitioner where we assume the block ID is already associated with each vertex. The subclass *MyWorker* of *BAssignWorker* is defined with its import/export UDF properly specified, and Line 17 is where the block information of a vertex is parsed into the vertex object.

[10] http://www.cse.cuhk.edu.hk/blogel/code/apps/block/sssp/STR.zip.

[11] http://www.cse.cuhk.edu.hk/blogel/code/apps/block/pagerank/urlPart.zip.

5.3.2 Block-Centric API

To write a block-centric program, a user needs to subclass the *BVertex*, *Block* and *BWorker* classes, which we introduce next.

The *BVertex* Class This is the vertex base class of the block-centric framework, and it is defined in the header file “blogel/BVertex.h”. There are three template arguments for users to specify: (1) *KeyT* is the type of vertex ID, (2) *ValueT* is the type of vertex value, and (3) *MessageT* is the type of messages received by vertices. There is no *HashT* like in the *Vertex* base class of Pregel+, since partitioner already determined the vertex-to-worker mapping, and the worker that a vertex v resides in is already stored in v’s triplet ID.

A *BVertex* object v maintains five fields: (1) vertex ID *id* (at Line 14), (2) v’s block ID *bid* (at Line 15), (3) v’s worker ID *wid* (at Line 16), (4) vertex value *_value* (at Line 82), and (5) boolean variable *active* (at Line 83) indicating whether the vertex is active. Similar to Pregel’s *Vertex* class, there is an abstract function *compute*(*messages*) for users to implement their vertex-centric algorithms, in which users can call functions like *vote_to_halt*() and *send_message*(.).

The difference from Pregel’s *Vertex* class is that, a *BVertex* vertex maintains its block ID *bid* and worker ID *wid*, and function *send_message*(*tgt*, *tgt_wid*, *msg*) requires users to specify target vertex by not only providing its vertex ID *tgt*, but also its worker ID *tgt_wid*. This is, however, straightforward since for a vertex v, its neighbor u’s worker ID is stored in u’s triplet ID, which can be easily accessed from v’s adjacency list.

The *Block* Class This is the block base class, and it is defined in the header file “blogel/Block.h”. There are three template arguments for users to specify: (1) *BValT* is the type of a block’s value field, (2) *BVertexT* is the type of the vertices that the block contains, which should be a subclass of *BVertex*, (3) *BMsgT* is the type of messages received by blocks.

A *Block* object b maintains five fields: (1) block ID *bid* (at Line 11), (2) beginning position of b’s vertex subarray in the vertex array, *begin* (at Line 12), which is illustrated in Fig. 5.1, (3) the length of b’s vertex subarray, *size* (at Line 13), which is also illustrated in Fig. 5.1, (4) b’s value *_value* (at Line 61), and (5) boolean variable *active* (at Line 62) indicating whether b is active.

Similar to the vertex base class, there is an abstract function *b.compute*(*messages*, *vertexes*) for users to implement a block’s algorithms for processing received messages, where *vertexes* is the vertex array of b’s worker, and the input argument allows users to access b’s vertices *vertexes*[*b.begin*], $\cdots$, *vertexes*[*b.begin* + *b.size* − 1]. In *compute*(.), a block can also call functions like *vote_to_halt*() and *send_message*(*tgt*, *tgt_wid*, *msg*), where *tgt* is the target block’s ID, and *tgt_wid* is the target block’s worker ID.

The *BWorker* Class This is the worker base class of the block-centric framework, and it is defined in the header file “blogel/BWorker.h”. It takes the user-defined *Block* subclass as an argument, and has three abstract functions for users to specify:

```
VertexT* toVertex(char* line)
void toline(BlockT* b, VertexT* v, BufferedWriter& writer)
//for dumping v's information, "b" is the block of "v"
void blockInit(vector& vertexList, vector& blockList)
//for initializing each block object using its vertices
```

The first two functions are import and export UDFs. Function *toline*(.) is called on every vertex v, and b is v's block which is also part of the function input so that v can output information about its block. The third function *blockInit*(.) is used to initialize each block object using its vertices, where *vertexList* and *blockList* are the vertex array and block array of the current worker, respectively. After the vertices are loaded, each worker will automatically construct its block array from the vertex array, and set the blocks' fields like *bid*, *start*, *size*. However, users need to initialize fields like the value of a block using *blockInit*(.), which is called before graph computation begins.

Before calling *run*(.) to start the computation, users need to specify the running mode using the following function, and the default setting of which is B-mode.

```
void set_compute_mode(int mode)
//mode = BWorker::B_COMP, or BWorker::V_COMP, or BWorker::VB_COMP
```

There are two kinds of messages: vertex-wise messages and block-wise messages. Accordingly one may call the following two functions to specify combiners for these two kinds of messages.

```
void setCombiner(Combiner* cb)     //vertex-wise message combiner
void setBCombiner(Combiner* cb)     //block-wise message combiner
```

BWorker also admits an optional template arguments *AggregatorT*, which should be a subclass of *BAggregator* defined in "blogel/BAggregator.h". Please refer to Blogel's *Architecture Overview* page[12] for the usage of combiners and aggregators.

Example We illustrate how to write a block-centric program, using the application code of computing single-source shortest paths.[13] File "blogel_app_sssp.h" defines the subclass *SPVertex* (Line 95) of *BVertex*, the subclass *SPBlock* (Line 139) of *Block*, the subclass *SPBlockWorker* (Line 230) of *BWorker*. The algorithm runs in VB-mode (see Line 355), but only vertex-wise messages are transmitted.

An *SPVertex* vertex has a value of type *SPValue* (Line 44), which maintains fields including (1) *dist*, the estimated distance from the source vertex, (2) *from*, the vertex of the previous hop on the current shortest path that generates the estimated distance, (3) *edges*, the list of neighbors each of type *SPEdge* (defined at Line 16, keeping a neighbor's vertex ID, block ID and worker ID, in addition to edge length).

[12]http://www.cse.cuhk.edu.hk/blogel/overview.html.

[13]http://www.cse.cuhk.edu.hk/blogel/code/apps/block/sssp/block.zip.

The type of vertex-wise message is *SPMsg*, which includes (1) *dist*, the estimated distance to the target vertex, and (2) the vertex of the previous hop on the current shortest path that generates the estimated distance.

In a superstep, vertices first call *SPVertex*::*compute*(.) (Line 98) to receive messages. If a smaller distance estimate is received by v, v updates its fields and remains active to become a source vertex for propagating new distance estimate to other vertices in its block (see Lines 126–130).

Then, every block calls *SPBlock*::*compute*(.) (Line 144) to propagate new distance estimates throughout its vertices. This is achieved by running Dijkstra's algorithm over the vertices in the block, and a min-heap *hp* (Line 148) is used to collect all active vertices in the block to initiate the execution of Dijkstra's algorithm.

References

1. G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.
2. S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.
3. Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.
4. W. Xie, G. Wang, D. Bindel, A. J. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14):2014–2025, 2013.
5. D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

Chapter 6
Subgraph-Centric Graph Mining

6.1 Problem Definition and Existing Methods

Problem Definition We focus on a class of graph mining problems, namely *subgraph finding* problems, which aim to find all subgraphs in a graph that satisfy certain requirements. It may enumerate (or count) all of the subgraphs, or find only those subgraphs with top-k highest scores, or simply output the largest subgraph. Examples of *subgraph finding* problems include *graph matching* [6], *maximum clique finding* [12], *maximal clique enumeration* [1], *quasi-clique enumeration* [7], *triangle listing and counting* [3], and *densest subgraph finding* [5]. These problems have a wide range of applications including social network analysis [8, 10], searching knowledge bases [4, 13] and biological network investigation [2, 14].

A common feature of subgraph finding problems is that, the computation over a graph G can be decomposed into that over subgraphs of G that are often much smaller (called **decomposed subgraphs**), such that each result subgraph is found in exactly one decomposed subgraph. In other words, the decomposed subgraphs partition the search space and there is no redundant computation. We illustrate by considering maximal clique enumeration, which serves as our running example. Table 6.1 summarizes the notations used throughout this chapter.

Example: Maximal Clique Enumeration We decompose a graph $G = (V, E)$ into a set of G's subgraphs $\{G_1, G_2, \ldots, G_n\}$, where G_i is constructed by expanding from a vertex $v_i \in V$. Let us denote the neighbors of a vertex v by $\Gamma(v)$. If we construct G_i as the subgraph induced by $\{v_i\} \cup \Gamma(v_i)$ (G_i is called v_i's 1-ego network), then we can find all cliques from these 1-ego networks since any two vertices in a clique must be neighbors of each other. However, a clique could be double-counted.

Let us define $\Gamma_{gt}(v) = \{u \in \Gamma(v) \mid u > v\}$ where vertices are compared according to their IDs. To avoid redundant computation, we redefine G_i as induced by $\{v_i\} \cup \Gamma_{gt}(v_i)$, i.e., G_i does not contain any neighbor $v_j < v_i$. This is because any

D. Yan et al., *Systems for Big Graph Analytics*, SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-58217-7_6

Table 6.1 Notation table

Notation	Meaning
$G = (V, E)$	G is the input graph, with vertex (edge) set $V(E)$
v_i	The i-th vertex of G
G_i	A decomposed subgraph of G seeded from v_i
$\Gamma(v)$	The set of neighboring vertices of v
$\Gamma_{gt}(v)$	Neighbors of v whose vertex IDs are larger than v's
$\mathbb{W}$	The set of all workers
$\|\mathbb{W}\|$	The total number of workers
W_i	The i-th worker
T_{local}	The local vertex table of a worker
$v^j{}_i$	The i-th vertex in T_{local} of worker W_j
T_{cache}	LRU cache of a worker to keep remote vertices
$\max_{\Gamma}(v)$	The vertex in $\Gamma(v)$ with the largest ID
$P(t)$	Vertices that task t needs to pull from remote machines

clique containing both v_i and v_j has already been computed when processing G_j. Obviously, any clique C (let the smallest vertex in C be v_i) is only computed once, i.e., when G_i is processed. □

As an illustration of the idea of subgraph-centric computation, we can distribute these decomposed subgraphs to different machines, so that each decomposed subgraph is processed using a serial backtracking algorithm to find cliques without network communication. Since the computation complexity of maximal clique enumeration is exponential to graph size, the computation cost of processing G_i is super-linear (to G_i's size) with a small constant (i.e., computation-intensive), while the transmission cost of creating G_i is linear with a large constant (due to limited network transmission rate). Thus, the computation cost and communication cost strike a balance when G_i is sufficiently large, and overlapping computation and communication over decomposed subgraphs significantly improves the overall performance.

However, since real graphs often follow power-law degree distribution, there may exist some vertex v_i with a very high degree, thus generating a large G_i. Due to high computational complexity, the machine processing G_i may becomes the straggler that keeps processing G_i while other machines finish their tasks and become idle. To tackle this problem, a system should allow G_i to be further decomposed, so that the resulting decomposed subgraphs can be distributed to different machines for processing. In maximal clique enumeration, we can decompose G_i exactly as how we decompose G, conditioned on that v_i is already in any clique found therein. The decomposition may recurse by looking at more vertices until the resulting subgraphs are small enough for balanced workload distribution.

Existing Solutions It is non-trivial to extend serial algorithms of subgraph finding problems for parallel processing, since a serial algorithm checks subgraphs using backtracking, where only one candidate subgraph is constructed (incrementally from the previous candidate subgraph) and examined at a time; a parallel algorithm that checks subgraphs simultaneously in memory many cause memory overflow, since the cumulative volume of all candidate subgraphs (that may overlap with each other) can be much larger than the input graph itself. Even worse, a big graph per se may not even fit into the memory of a machine.

The vertex-centric framework presented in Part I is not a good solution to subgraph finding, since vertex-centric programs are mostly data-intensive. Specifically, the processing of each vertex is triggered by incoming messages sent (mostly) from other machines, and the CPU cost of vertex processing is negligible compared with the communication cost of message transmission. Moreover, subgraph finding problems operate on subgraphs rather than individual vertices, and it is unnatural to translate a subgraph finding problem into a vertex-centric program. Each vertex v_i needs to communicate with its surrounding vertices in a breadth-first manner (one more hop per iteration) to get their information for constructing G_i. A vertex-centric system with out-of-core is also essential, since it is prohibitive to keep all decomposed subgraphs in memory.

The block-centric framework presented in the previous chapter is just to accelerate vertex-centric models by eliminating the need of communication inside each block. It partitions a graph into *disjoint* subgraphs (i.e., blocks) rather than overlapping ones, and the computation model is still iterative and requires global synchronization among all machines.

While dominating research efforts on big graph systems have been devoted to problems that naturally have a vertex-program implementation, a couple of recent systems attempt to develop subgraph-centric frameworks that are better suited to mine a big graph, including NScale [9], Arabesque [11] and *G-thinker*.[1] Unfortunately, NScale and Arabesque still mine subgraphs in a data-intensive manner: candidate subgraphs are first constructed synchronously from small ones to large ones, before the actual computation-intensive mining process. In contrast, *G-thinker* is able to handle the computation-intensive workloads. We briefly review NScale and Arabesque next, and will describe *G-thinker* in detail in the next section.

NScale [9] only supports the top-level decomposed subgraphs, and there is no mechanism to balance workload through recursive decomposition. Assuming that each decomposed subgraph G_i spans the k-hop neighborhood around each vertex v_i, then NScale first constructs all decomposed subgraphs using k rounds of MapReduce. The large number of decomposed subgraphs are then packed into larger compact subgraphs, each of which can fit in the memory of a reducer. Vertices common to multiple decomposed subgraphs are stored only once in their packed subgraph. Finally, each compact subgraph is distributed to a reducer, which processes all decomposed subgraphs packed in the compact subgraph in memory. Obviously, NScale suffers from all the performance issues of a vertex-centric solution, as well as the huge overhead of repeated HDFS data loading/dumping. NScale also brings new overhead. For example, NScale packs decomposed subgraphs

[1] http:///www.cis.uab.edu/yanda/gthinker.

through expensive disk-based computation, and it is very likely that the cost of packing G_i already surpasses that of processing G_i right after it is constructed in memory.

Arabesque [11] proposed an embedding-centric model where an embedding is a subgraph of the input graph G. Arabesque requires the entire G to reside in the memory of every machine, and constructs and processes subgraphs iteratively. In the i-th iteration, it grows the set of embeddings with i edges/vertices by one adjacent edge/vertex, to construct embeddings with $(i + 1)$ edges/vertices for processing. New embeddings that pass a filtering condition are further processed and then passed to the next iteration. For example, to find cliques, the filtering condition checks whether an embedding e is a clique; if so, e is reported and passed to the next iteration to grow larger clique candidates. Unfortunately, Arabesque suffers from new performance and scalability issues. Firstly, while previous solutions still permit efficient backtracking within each decomposed subgraph, Arabesque materializes and transmits every single candidate subgraph it examines (e.g., clique in the example above). Arabesque also compresses/decompresses the large number of materialized embeddings using a data structure called ODAG to save space, which consumes additional CPU cycles. To additionally support frequent subgraph pattern mining, automorphism checking is performed for every newly-expanded embedding to avoid generating duplicate embeddings, which adds unnecessary overhead for subgraph finding. Finally, since G resides in the memory of every machine, scalability is limited by the memory space of a single machine.

6.2 The G-Thinker System

Overview We identify the following five requirements that a distributed system for subgraph finding should satisfy in order to be efficient and user-friendly:

- The programming interface should be **subgraph-centric**.
- **Computation-intensive** processing should be native to the programming model. For example, a programmer should be able to backtrack a portion of graph to examine candidate subgraphs like in a serial algorithm, without materializing and transmitting any subgraph.
- There should exist **no global synchronization** among the machines, i.e., the processing of different portions of a graph should not block each other.
- Since a subgraph finding algorithm checks many (possibly overlapping) subgraphs whose cumulative volume can be much larger than the input graph itself, it is important to schedule the subgraph-tasks properly to keep the **memory usage bounded** at any point of time. Obviously, each machine should stream and process its subgraphs on its local disk (if memory is not sufficient), in order to minimize network and disk IO overhead.
- Subgraphs on a machine that contain a common vertex v should be able to share v's information (e.g., adjacent edges), to **avoid redundant data transmission and storage**. This is not the case in existing distributed frameworks which

process each individual subgraph as an independent task: if multiple subgraphs on a machine contain a common vertex v, each subgraph will maintain its own copy of v's information (likely received from another machine).

G-thinker, designed based on the above criteria, is a unified framework for developing scalable algorithms for various subgraph fining problems. To write a G-thinker program, a user only needs to specify how to grow a portion of the input graph g by pulling g's surrounding vertices, and how to process g (e.g., by backtracking). Communication and execution details in G-thinker are transparent to end users. In G-thinker, each machine only keeps and processes a small batch of tasks in memory at any time. Task batching helps to achieve high throughput while keeping memory usage bounded. Subgraphs that are waiting to be processed (e.g., to grow its frontier by pulling remote vertices) are buffered in a disk-based queue. Experiments show that G-thinker is up to hundreds of times faster and scales to graphs that are two orders of magnitude larger given the same hardware resources.

Programming Model G-thinker performs computation on subgraphs. Each subgraph g is associated with a ***task***, which performs computation on g and grows g when needed. G-thinker grows subgraphs starting from a set of seed vertices in V. For example, in clique enumeration, one may create a task from each vertex $v_i \in V$, which forms the initial subgraph g containing only v_i; the task grows g into G_i by ***pulling*** vertices in $\Gamma_{gt}(v_i)$ along with their adjacent edges, and then enumerates cliques in G_i. In case G_i is too big, users may instead further decompose G_i and create new tasks associated with the newly decomposed subgraphs, which can then be distributed to different machines to improve load balancing.

Components A G-thinker program runs on a cluster of workers, $\mathbb{W} = \{W_1, W_2, \ldots\}$, where each worker is a basic computing unit that processes its assigned tasks in serial, and a machine may run multiple workers. Each worker alternates between subgraph-centric task computation and vertex pulling (into subgraphs), both are processed in batches. To be memory efficient, we keep the memory requirement of each worker at approximately $O(d_{avg} \cdot \frac{|V|}{|\mathbb{W}|})$, where d_{avg} is the average vertex degree.

G-thinker partitions the vertices in V (along with their adjacency lists) among different workers, and each worker maintains its assigned vertices in a local table T_{local}. Let us denote the i-th vertex maintained in T_{local} of worker W_j by v_i^j. Figure 6.1 shows the components of W_1, where we can see that T_{local} maintains

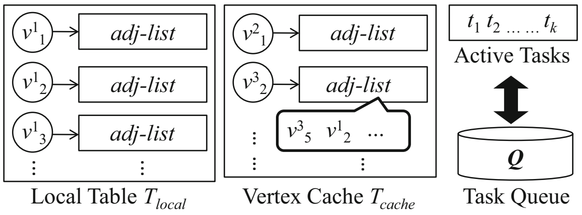


Fig. 6.1 Components of worker W_1

vertices $v_1^1, v_2^1, v_3^1, \ldots$; each vertex also keeps its neighbors in its adjacency list, so that it can pull its neighbors (along with their adjacency lists) from T_{local} of other workers, by providing the neighbor IDs. The local tables of all workers collectively constitute a distributed key-value store where key is the ID of a vertex v and value is v's adjacency list $\Gamma(v)$.

When a vertex is pulled (along with its adjacency list) from another worker, it is not directly added to the subgraph of the requesting task; instead, it is put in an *LRU cache* T_{cache}. The cache keeps the non-local vertices (i.e., not in T_{local}) that are previously received, so that a non-local vertex can be shared by all the tasks that pull it. It is up to the user to decide whether the task's subgraph should be updated, and if so, what information of that vertex should be added to the subgraph.

As Fig. 6.1 shows, W_1's T_{cache} keeps non-local vertices $v_1^2, v_2^3, \ldots$ (along with their adjacency lists), which are pulled by local tasks previously executed at W_1. Note that the adjacency list of a non-local vertex may contain a local vertex, such as v_2^1 in the adjacency list of vertex v_2^3 in T_{cache} in Fig. 6.1.

During subgraph-centric computation, when a task requires the adjacency list of a vertex u, if u is in T_{local} or T_{cache}, the task can directly obtain $\Gamma(u)$ by table lookup. Otherwise, the task needs to first pull u from T_{local} of u's worker into the cache table T_{cache}, before accessing it.

As Fig. 6.1 shows, each worker also maintains an *in-memory task buffer* for keeping tasks that are currently being processed, and a *disk-based task queue* for keeping tasks that are waiting to be processed. This design allows tasks to be processed with high throughput and less redundant communication, as we shall discuss next.

Batch Processing and Communication Reduction A task usually only generates a small number of pull-requests at a time, and sending small messages wastes network bandwidth. Therefore, in G-thinker, a worker fetches a batch of tasks from the disk-based task queue at each time, sends their pull-requests together, receives all the requested vertices, and then processes these tasks. Tasks that need to pull more vertices are then added to the task queue for further processing.

Batch processing hides the round-trip delay of each task's pull-requests, since if tasks are processed one at a time, each task needs to wait for its requested vertices to arrive, which wastes CPU cycles. Batch processing also reduces redundancy in communication. Specifically, if many tasks in a batch pull a remote vertex u, only one pull-request needs to be sent, and $\langle u, \Gamma(u)\rangle$ will be received only once and cached in T_{cache} for access by all these tasks. To further reduce communication, G-thinker allows a user to prune useless items in $\Gamma(v)$ before responding $\langle v, \Gamma(v)\rangle$ to a worker that pulls v. For example, in clique enumeration, a vertex v only needs to respond to a pull-request with $\Gamma_{gt}(v)$ instead of the entire $\Gamma(v)$.

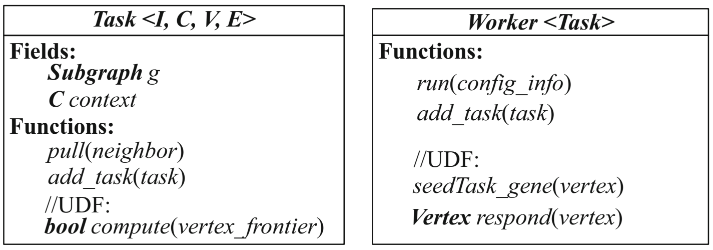
Computation and Communication Cost Analysis Each task in G-thinker (1) pulls required vertices to its subgraph (linear communication cost, and pull requests can further be shared with other tasks) and (2) then performs higher-complexity computation on the subgraph in local machine. Step (2) is computation-intensive and avoids any communication when exploring the large search space by backtracking.

To overlap communication (i.e., Step (1)) with computation (i.e., Step (2)), G-thinker treats tasks independently. Different tasks can have different progress, and no synchronization among all machines is required. G-thinker proceeds the computation of a task as long as all its requested vertices are locally accessible, and the only communication type in G-thinker is point-to-point communication between two workers for vertex pulling, and dynamic task (or decomposed subgraph) reassignment if load balancing is enabled.

Programming Interface G-thinker is written in C++, and it defines two important base classes, *Task* and *Worker*, as sketched in Fig. 6.2. To write a G-thinker program, a user needs to subclass *Task* and *Worker* with their template arguments properly specified, and to implement their abstract functions according to the application logic.

Data Types As Fig. 6.2 shows, the *Task* class takes four template arguments $< I >$, $< C >$, $< V >$ and $< E >$. Among them, $< I >$, $< V >$ and $< E >$ specify the data types of vertices and edges: (1) $< I >$: the type of vertex ID; (2) $< V >$: the type of vertex attribute; (3) $< E >$: the type of the attribute of an adjacency list item. Other system-defined types (e.g., those for subgraph, vertex, and adjacency list) are automatically derived by *G-thinker* from them, and can be directly used in the UDFs once a user specifies these three template arguments.

Figure 6.3 illustrates the inferred system-defined types. Specifically, a subgraph is shown on the left, which consists of a table of vertices (stored with their adjacency lists). The structure of Vertex 3 is shown on the right, where the vertex is stored with its ID (of type $< I >$) and a vertex label "c" (of type $< V >$), and an adjacency list. Each item in the adjacency list is stored with a neighbor ID (of type $< I >$) and an attribute (of type $< E >$) indicating the label of the neighbor and the edge label. For example, the first item corresponds to Vertex 1 with label "a", and the edge label of

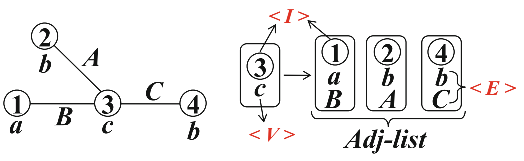


Task <I, C, V, E>
Fields:
    Subgraph *g*
    C *context*
Functions:
    pull(*neighbor*)
    add_task(*task*)
    //UDF:
    bool *compute*(*vertex_frontier*)

Worker <Task>
Functions:
    run(*config_info*)
    add_task(*task*)
    //UDF:
    seedTask_gene(*vertex*)
    Vertex *respond*(*vertex*)

Fig. 6.2 Programming interface of G-thinker

Fig. 6.3 Data types in G-thinker



$(3, 1)$ is "B". Attributes (i.e., $< V >$ and $< E >$) are optional and are not needed for finding subgraphs with only topology constraints (e.g., triangles, cliques, and quasi-cliques).

The Task Class The *Task* class has another template argument $< C >$ that specifies the type of context information for a task *t*, which can be, for example, *t*'s iteration number (a task in G-thinker proceeds its computation in iterations). Each *Task* object *t* maintains a subgraph *g* and the user-specified *context* object (of type $< C >$). The *Task* class has only one UDF, *t.compute*(*frontier*), where the input *frontier* keeps the set of vertices requested by *t* in its previous iteration. Each element of *frontier* is actually a pointer to a vertex object in T_{local} or T_{cache}. Of course, users may also access *t*'s subgraph and context object in *compute*(*frontier*).

UDF *compute*(*frontier*) specifies how a task computes for one iteration. If *t.compute*(.) returns *true*, *t* needs to be processed by more iterations; otherwise, *t*'s computation is finished after the current iteration. In G-thinker, when *t* is fetched from the task queue for processing, *t.compute*(.) is executed repeatedly until either *t* is complete, or *t* needs a non-local vertex v that is not cached in T_{cache}, in which case *t* is added to the task queue waiting for all *t*'s requested vertices to be pulled. When a task is completed or queued to disk, G-thinker automatically garbage collects the memory space of the task to make room for the processing of other tasks.

Inside *t.compute*(.), a user may access and update *g* and *context*, and call *pull*(*u*) to request vertex *u* for use in *t*'s next iteration. Here, *u* is usually in the adjacency list of a previously pulled vertex, and *pull*(*u*) expands the frontier of *g*. To improve network utilization, *g* is usually expanded in a breadth-first manner, so that each call of *compute*(.) generates pull-requests for all relevant vertices adjacent to *g*'s growing frontier. A user may also call *add_task*(*task*) in *t.compute*(.) to add a newly-created task to the task queue.

The Worker Class Each object of the *Worker* class corresponds to a worker that processes its assigned tasks in serial. Figure 6.2 shows the key functions of the *Worker* class, including two important UDFs.

UDF *seedTask_gene*(v) specifies how to create tasks according to a seed vertex $v \in T_{local}$. A worker of G-thinker starts by calling *seedTask_gene*(v) on every $v \in T_{local}$, to generate seed tasks and to add them to the disk-based task queue. Inside *seedTask_gene*(v), users may examine the adjacency list of v, create tasks accordingly (and may let each task pull neighbors of v), and add these tasks to the task queue by calling *add_task(.)*.

UDF *respond*(v) is used to prune $\Gamma(v)$ before sending it back to requesting workers. By default, *respond*(v) returns *NULL* and G-thinker directly uses the vertex object of v in T_{local} to respond. Users may overload *respond*(v) to return a newly created copy of v, with items in $\Gamma(v)$ properly pruned to save communication (e.g., $\Gamma_{gt}(v)$ for clique enumeration). In this case, G-thinker will respond by sending the new object and then garbage-collect it.

The *worker* class also contains formatting UDFs, e.g., for users to define how to parse a line in the input file on HDFS into a vertex object in T_{local}, which will be used during graph loading.

To run a G-thinker program, one may subclass *Worker* with all UDFs properly implemented, and then call *run*(*config_info*) to start the job, where *config_info* contains job configuration parameters such as the HDFS file path of the input graph.

The Aggregator Class The *Worker* class optionally admits a second template argument <*aggT*>, which needs to be specified if aggregator is used to collect some statistics such as triangle count or maximum clique size. Each task can aggregate a value to its worker's local aggregator when it finishes. These locally aggregated values can either be globally aggregated at last when all workers finish computing their tasks (which is the default setting), or be periodically synchronized (e.g., every 10 s) to make the globally aggregated value available to all workers (and thus all tasks) timely for use (e.g., in *compute*(.) to prune search space). In the latter case, users need to provide a frequency parameter.

Application Example: Maximum Clique We illustrate how to write a G-thinker program by considering the maximum clique problem. We adapt the serial back-tracking algorithm of [12] to G-thinker. The original algorithm maintains the size of the maximum clique currently found, denoted by $|Q_{max}|$, to prune the search space.

To allow timely pruning, each worker in our G-thinker program maintains $|Q_{max}|$ and keeps it relatively up to date by periodic aggregator synchronization, so that if a worker discovers a larger clique and updates $|Q_{max}|$, the value can be synchronized to other workers timely to improve their pruning effectiveness. In *seedTask_gene*(v_i), we create a task t whose graph g contains v_i, and we let t pull all vertices in $\Gamma_{gt}(v_i)$. Then in *t.compute*(*frontier*), we collect vertices in *frontier* (i.e., $\Gamma_{gt}(v_i)$), add them to g but filter those adjacency list items that are not in $\{v_i\} \cup \Gamma_{gt}(v_i)$, to form the decomposed subgraph G_i, and then run the algorithm of [12] on G_i.

This solution can be easily extended to find quasi-cliques, where in a quasi-clique, every vertex is adjacent to at least γ (≥ 0.5) fraction of other vertices. In such a quasi-clique, two vertices are at most two hops away [7]. The G-thinker algorithm is similar to that for finding maximum clique, except that (1) for each local seeding vertex v_i, *compute*(*frontier*) runs for two iterations to pull vertices (larger than v_i) within two hops of v_i; (2) *compute*(*frontier*) then constructs G_i as the two-hop ego-network of v_i and runs the quasi-clique algorithm of [7] on G_i to compute the quasi-cliques.

Get Started Using G-Thinker *G-thinker*'s website[2] provides the system and application code of G-thinker, and detailed documentations to help you get started. The current system version requires you to first partition the input graph among different workers, before performing distributed subgraph-centric processing. Multiple partitioners are provided under the library (i.e., folder) "partition",[3] while the subgraph-centric API is under the "subgraph" library.

[2]http:///www.cis.uab.edu/yanda/gthinker.

[3]You may simply use the hash-partitioner that distributes vertices to workers by hashing vertex ID. Please refer to an example application code for its usage.

References

1. C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.
2. H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418, 2008.
3. X. Hu, Y. Tao, and C. Chung. I/o-efficient algorithms on triangle listing and counting. *ACM Trans. Database Syst.*, 39(4):27:1–27:30, 2014.
4. G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.
5. S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, pages 597–608, 2009.
6. J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.
7. G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In *ECML/PKDD Part II*, pages 33–49, 2008.
8. J. Pattillo, N. Youssef, and S. Butenko. On clique relaxation models in network analysis. *European Journal of Operational Research*, 226(1):9–18, 2013.
9. A. Quamar, A. Deshpande, and J. Lin. NScale: neighborhood-centric large-scale graph analytics in the cloud. *VLDB Journal*, 25(2):125–150, 2016.
10. L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463, 2012.
11. C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.
12. E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *DMTCS*, pages 278–289, 2003.
13. W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, pages 481–492, 2012.
14. L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.

Part III
Think Like a Matrix

Chapter 7
Matrix-Based Graph Systems

7.1 Graphs and Matrices

Graphs and matrices are inherently related: a graph can be represented by either an adjacency matrix or an incidence matrix, and operations on graphs can be performed by algebraic operations on matrices. As a result, linear algebra and matrix theory can be applied to solve graph problems. Algebraic graph theory is a dedicated field to study algebraic methods for solving graph problems. For interested readers, we refer to books by Biggs [1] and by Godsil and Royle [7] for in-depth study in this topic.

Adjacency Matrix An *adjacency matrix* can be used to represent a directed or undirected *simple* graph, i.e. a graph without cycles and parallel edges. Given a graph $G = (V, E)$, its adjacency matrix $A(G)$, or simply A if unambiguous, is a $|V| \times |V|$ matrix in which each element $A[i][j]$ represents whether there is an edge connecting the ith vertex (denoted as v_i) and the jth vertex (denoted as v_j) in V. The value of $A[i][j]$ is defined as follows:

$$A[i][j] = \begin{cases} 1 & \text{if } i \neq j \text{ and } (v_i, v_j) \in E \\ 0 & \text{if } i \neq j \text{ and } (v_i, v_j) \notin E \\ 0 & \text{if } i = j \end{cases}$$

Incidence Matrix An *incidence matrix* is another way to represent a graph. Given a Graph $G = (V, E)$, its incident matrix $B(G)$, or simply B if there is no ambiguity, is a $|V| \times |E|$ matrix, where each element $B[i][j]$ denotes whether the ith vertex in V (denoted as v_i) and the jth edge in E (denoted as e_j) are incident. More formally, in terms of undirected graph,

$$B[i][j] = \begin{cases} 1 & \text{if } v_i \text{ and } e_j \text{ are incident} \\ 0 & \textit{otherwise} \end{cases}$$

D. Yan et al., *Systems for Big Graph Analytics*, SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-58217-7_7

and in terms of directed graph,

$$B[i][j] = \begin{cases} 1 & \text{if } e_j \text{ originates at } v_i \\ -1 & \text{if } e_j \text{ terminates at } v_i \\ 0 & \textit{otherwise} \end{cases}$$

With the matrix representation of graphs, many graph operations can be translated into linear algebra operations on the matrices. For example, we can produce a vector representing the in-neighbors (or out-neighbors) of a vertex v_i by computing $A \cdot \mathbf{x}_{v_i}$ (or $A^T \cdot \mathbf{x}_{v_i}$), where $\mathbf{x}_{v_i}$ is the column ($|V| \times 1$) indicator vector of v_i (i.e., $\mathbf{x}_{v_i}[i] = 1$ and all other elements of $\mathbf{x}_{v_i}$ are 0). As another example, to compute the subgraph induced by a subset of vertices S of an undirected graph, we can compute $\mathbf{x}_S \cdot B$, where $\mathbf{x}_S$ is the row ($1 \times |V|$) indicator vector for S, (i.e. $\mathbf{x}_S[i] = 1$ if $v_i \in S$ and $\mathbf{x}_S[i] = 0$ otherwise. The result vector consists of elements with 0, 1, or 2, and the induced subgraph is given by those edges whose corresponding values are 2 (since it means that both endpoints of the edges are in S).

All the systems introduced in this chapter exploit the inherent connection between graphs and matrices, and expose a matrix-based interface for users to express their graph problems.

7.2 PEGASUS

PEGASUS [11, 12] is a graph system developed by U Kang et al from Carnegie Mellon University. Although it was open source at http://www.cs.cmu.edu/~pegasus, there hasn't been any change to the code base since 2010.

PEGASUS was among the very first wave of general-purpose big graph systems, even before the advent of Pregel. Before PEGASUS, big graphs were processed using MapReduce, and many tailor-made MapReduce algorithms were proposed for solving specific graph problems. Kang et al. pioneered PEGASUS, which is a general-purpose MapReduce-based framework for designing parallel graph algorithms, based on the observation that many graph computation can be modeled by a generalized form of matrix-vector multiplication.

7.2.1 Programming Model

PEGASUS targets at iterative graph algorithms. It models each iteration of the graph computation by a generalized matrix-vector multiplication operation, which is repeated until the vertex values in the vector converge. Taking the PageRank algorithm as an example, each iteration can be formulated by the following matrix-vector multiplication operation:

$$\mathbf{v} \leftarrow (0.85 \cdot A^T + 0.15 \cdot U) \cdot \mathbf{v}. \tag{7.1}$$

In Eq. (7.1), $\mathbf{v}$ is a column vector with $|V|$ elements, and each element $\mathbf{v}[i]$ refers to the value of a vertex v_i (i.e., $a(v_i)$); A is a modified adjacency matrix, and $A[i][j] = 1/d_{out}(v_i)$ if edge (v_i, v_j) exists, and $A[i][j] = 0$ otherwise; U is a $|V| \times |V|$ matrix with all elements set to $1/|V|$. Let us denote $M = 0.85 \cdot A^T + 0.15 \cdot U$, then Eq. (7.1) can be written as a matrix-vector multiplication $\mathbf{v} \leftarrow M \cdot \mathbf{v}$.

In PEGASUS, the generalized matrix-vector multiplication is defined by three operators:

1. combine2: to combine $M[i][j]$ and $\mathbf{v}[j]$ into a value, e.g. $M[i][j] \cdot \mathbf{v}[j]$;
2. combineAll: for each v_i, to combine all the $|V|$ intermediate results produced by combine2 into a single value, e.g., $\sum_{j=1}^{|V|}$ combine2$(M[i][j], \mathbf{v}[j])$;
3. assign: to overwrite the old value of v_i with the new value.

In the PageRank algorithm, combine2 simply performs multiplication $M[i][j] \cdot \mathbf{v}[j]$ while combineAll simply performs summation. PEGASUS lets a user customize the above three operators to implement different graph algorithms. For example, to implement the Hash-Min algorithm, we simply set M to be the 0–1 adjacency matrix of an undirected graph G, and the three operators are defined as follows:

1. $combine2(i, j) = M[i][j] \cdot \mathbf{v}[j]$;
2. $combineAll(i) = \min_{j=1}^{|V|}\{combine2(i, j)\}$;
3. assign: $\mathbf{v}[i] \leftarrow \min\{\mathbf{v}[i], combineAll(i)\}$.

Note that for every iteration, PEGASUS processes all vertices in a graph. For example, in the above Hash-Min algorithm, even in later iterations when most vertices have already converged, every vertex v_i needs to recompute its value since there is no mechanism for individual vertices to halt.

7.2.2 System Implementation Overview

In PEGASUS, each iteration of graph computation is performed by two MapReduce jobs: The first job computes values of combine2(); The second groups the intermediate results by the vertex id, combines the results, and assigns the new value for each vertex.

To improve performance, PEGASUS partitions the matrix M into $b \times b$ submatrices, and partitions the vector $\mathbf{v}$ into groups of b elements, so that the matrix-vector multiplication is performed in coarser granularity. PEGASUS further adopts two optimizations to the block multiplication approach. Firstly, it preprocesses the matrix M by co-clustering (i.e. reordering rows and columns), so that there are fewer submatrices (a submatrix with all 0s are removed) but each submatrix contains more non-zero elements. Secondly, for some algorithms, such as Hash-Min, to reduce the number of iterations (i.e., MapReduce jobs), PEGASUS multiplies each diagonal matrix block with the corresponding vector block repeatedly in each iteration until the contents of the vector block do not change. This optimization propagates vertex states throughout the whole submatrix block (i.e., a subset of vertices), and thus reduces the number of iterations. This method shares the same idea as the block-centric computation.

7.3 GBASE

GBASE [9, 10] is another MapReduce-based big graph system. It is part of the IBM System G Toolkit (http://systemg.research.ibm.com).

Although GBASE also uses matrix-vector multiplication to perform graph computation, it has the following differences from PEGASUS. Firstly, GBASE aims to efficiently support "targeted" queries which need to access only part of the graph, in addition to "global" queries that traverse the whole graph. Secondly, a user of GBASE uses built-in graph operations to process and mine graphs, rather than write their own algorithms. Thirdly, each built-in graph operation is implemented by the exact matrix-vector multiplication(s) (run on MapReduce), not the generalized matrix-vector multiplication(s).

7.3.1 Basic Graph Operations

GBASE unifies different graph operations by the matrix-vector multiplication operation $\mathbf{v}_{out} \leftarrow M \cdot \mathbf{v}_{in}$. Here, $\mathbf{v}_{in}$ is a column vector with $|V|$ elements (one for each vertex), which serves as the query input. There are two cases for M and $\mathbf{v}_{out}$, which we describe below.

Case 1 M is the adjacency matrix A or its transpose A^T, and $\mathbf{v}_{out}$ is a column vector with $|V|$ elements (one for each vertex).

Case 2 M is the transpose of the incidence matrix B^T and $\mathbf{v}_{out}$ is a column vector with $|E|$ elements (one for each edge).

Then, these basic matrix-vector multiplication operations can be further combined to perform more advanced graph operations. For example, to get the set of vertices within k hops from v_i, one may left-multiply its indicator vector $\mathbf{x}_{v_i}$ by A^T for k times; while to get the k-ego network of v_i, one may multiply the result vector of v_i's k-hop neighbors with B.

7.3.2 Storing Matrices

To efficiently support the querying of a graph G, GBASE needs to first preprocess the adjacency matrix A into multiple files, so that during the processing of a query q, only those files relevant to q are read by MapReduce.

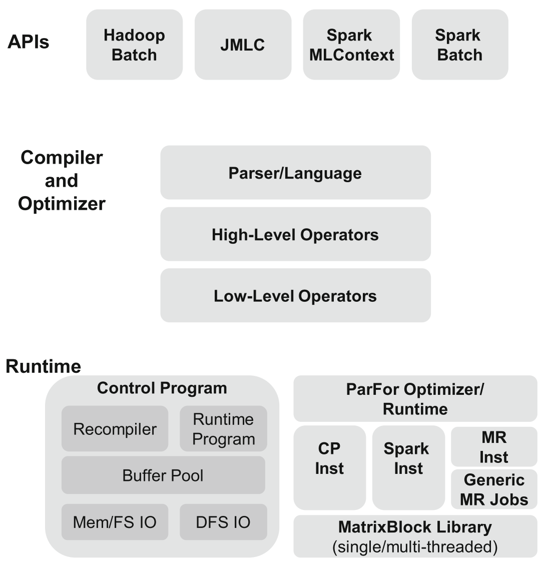
Similar to matrix co-clustering in PEGASUS, GBASE reorders the rows and columns of A and partitions A into homogeneous submatrices called blocks, with each block being either very dense or very sparse. GBASE then converts each block into a binary string (e.g., by concatenating the rows of the 0–1 submatrix), and then compresses the string using GZip. Kang et al. [9] reported that the total size of the compressed files is less than 2% of the original graph size. As a tradeoff, the files have to be unzipped for processing, but the storage savings and the reduced amount of data transfer outweigh the decompression overhead. To ensure that in-neighbor and out-neighbor queries are equally efficient, GBASE divides the matrix of blocks by a coarser-grained grid, and groups all blocks in each grid cell into a file. Let the total number of files be n, then with the grid placement of blocks, both an in-neighbor query and an out-neighbor query read $O(\sqrt{n})$ files.

7.4 SystemML

SystemML [6] started as a research project from IBM Research in 2010, and is now an very active Apache incubator project (https://systemml.apache.org). Since its open source in 2015 to the day of writing (March 14, 2017), there have been 41 contributors and 4436 commits to this project. For interested readers who want to try it out, the project website provides rich documentation on programming guild, language guide, algorithm references, debugger, IDE, trouble shooting, etc. Among all the three matrix-based systems introduced in this chapter, SystemML is certainly the best-maintained and most user-friendly one. Hence, we provide more detailed description to SystemML in this section.

Different from PEGASUS and GBASE, SystemML [6] takes a *declarative* approach to graph analytics and, more generally, to machine learning. Instead of exposing programming APIs to the end users, SystemML provides a high-level scripting language for users to express their analytics algorithms, without them worrying about how to execute individual operations (hence it is a declarative approach). SystemML compiles the algorithm scripts, and then optimizes and executes them in a hybrid runtime of a single node and a distributed cluster by using either MapReduce (MR) or Spark. Furthermore, SystemML supports general linear algebra operations, besides just matrix-vector multiplication, and is thus applicable for any machine learning or analytics algorithms that can be expressed using linear algebra. We now provide an overview of SystemML. Figure 7.1 pictorializes the overall architecture of SystemML.

Fig. 7.1 Architecture overview of SystemML



7.4.1 *Programming Language and User Interface*

The *declarative* approach that SystemML takes for machine learning (including graph analytics) is mainly achieved through its high-level **D**eclarative **M**achine learning **L**anguage (DML). There are two flavors to this language: one with an R-like syntax (DML) and one with a Python-like syntax (PyDML). This language includes linear algebra operations, statistical functions, control flow constructs like loops and branches, and user-defined functions. Script 1 shows how the same PageRank algorithm in Eq. (7.1) can be written in just nine lines of simple code in the R-like DML. As shown, this script is pretty much the direct translation of Eq. (7.1), thus is very intuitive to users, who are familiar with algorithm designs in terms of linear algebra. For more information on the language constructs, please refer to the DML language reference at http://apache.github.io/incubator-systemml/dml-language-reference.html.

Script 1 *PageRank Algorithm in R-like DML*

```
A = read("in/A", rows=1e6, cols=1e6, nnz=1e9); // A: the modified adjacency matrix, $A[i][j] = 1/d_{out}(v_i)$ if $(v_i, v_j) \in E$, otherwise $A[i][j] = 0$
v = read("in/v", rows=1e6, cols=1); //v: initial uniform pagerank
U = read("in/U", rows=1e6, cols=1e6); //U: a matrix with all elements set to $1/|V|$
max_iteration = 100;
i = 0;
while(i < max_iteration) {
v = (0.85 * t(A) + 0.15 * U) %*% p;
i = i + 1; }
write(v, "out/v");
```

SystemML provides many ways to run the analytics algorithms written in DML or PyDML. The standalone mode[1] is designed for rapid prototyping and testing of algorithms on a single machine. It is not appropriate for large datasets. For large datasets, one can then run the same script in the Hadoop Batch[2] or Spark Batch[3] modes, to take advantage of the distributed processing power of Hadoop or Spark, respectively. To make the interplay of SystemML and Spark more seamless, SystemML also provides a programmatic interface, MLContext API,[4] for Spark to easily interact with SystemML using Scala, Java, and Python. Using the MLContext, one can easily pass in Spark DataFrames or RDDs as input matrices to SystemML, and get result matrices from SystemML as DataFrames or RDDs in Spark as well. As a result, users can conveniently interact with SystemML from Spark Shell and Notebooks, such as Jupyter and Zeppelin. Finally, for scoring purposes (e.g. applying a learned model to real data for prediction), SystemML can also interact with the Java Machine Learning Connector (JMLC) API in an embedded fashion,[5] to support applying the scoring functions expressed in DML.

7.4.2 Compilation and Optimization

The high-level scripts written in DML or PyDML are dynamically compiled and optimized based on data and cluster characteristics using rule and cost-based optimization techniques. The compiler automatically generates hybrid runtime execution plans ranging from in-memory, single node execution to distributed MapReduce or Spark computation and data access. The cost-based optimizer in SystemML generates execution plans based on data characteristics, like the dimensionality and sparsity of matrices, as well as cluster and machine characteristics, such as memory and CPUs. We now provide a brief overview of the SystemML compiler and optimizer. For more details, please refer to [2].

Parser The scripts first go through the *parser* to perform lexical and syntactic analysis, live variable analysis, and machine learning specific validation (e.g. compatibility checks of dimensions for matrix operations).

High-Level Operators (HOPs) Next, the optimizer creates DAGs of *high-level operators* (HOPs) to represent the program structure. Nodes represent (logical) operations and their outputs (scalar/matrix with specific value type), while edges represent data dependencies. Rule-based rewrites that are independent of matrix

[1]http://apache.github.io/incubator-systemml/standalone-guide.html.

[2]http://apache.github.io/incubator-systemml/hadoop-batch-mode.html.

[3]http://apache.github.io/incubator-systemml/spark-batch-mode.html.

[4]http://apache.github.io/incubator-systemml/spark-mlcontext-programming-guide.html.

[5]http://apache.github.io/incubator-systemml/jmlc.html.

sizes, such as common subexpression elimination (CSE), constant folding, algebraic simplifications, and branch removal, are applied to transform the DAGs. After propagating the size information matrices across the program structure, another set of cost-based size-dependent rewrites, such as matrix multiplication chain optimization and dynamic algebraic simplifications, are applied to the HOP DAGs. Then based on the propagated sizes, the optimizer computes worst-case memory estimates for all HOPs.

Low-Level Operations (LOPs) In this step, SystemML's optimizer selects low-level operators (LOPs) for the HOPs, and transforms the HOP DAGs into LOP DAGs, where nodes represent (physical) operations and edges represent again data dependencies. The LOP selection is decided between a single-node runtime (called CP) and a distributed runtime (either MR or Spark), based on hard constraints w.r.t. memory estimates and budget as well as on specific physical operators. LOPs are runtime-backend-specific, but a DAG can contain mixed LOP types of integrated runtimes like CP and MR/Spark. Once the LOP DAGs are complete, the runtime program is generated, consisting of CP and MR/Spark instructions. If the distributed backend is MR, SystemML strives to minimize the number of MR jobs by greedily *piggybacking* multiple MR LOPs into composite MR jobs, based on constraints such as map/reduce execution location and memory constraints.

7.4.3 Runtime

Matrix Representation Similar to PEGASUS and GBASE, a matrix in SystemML is divided into blocks for efficient processing and storage. However, unlike the other two systems, SystemML does not require the expensive clustering of vertices in a preprocessing step to form the matrix blocks. At the per matrix-block level, dynamic runtime optimizations are also applied for choosing block layout (sparse or dense layout) and implementations of block-level operations, based on the statistics (e.g. density of a matrix block) gathered at runtime. A MatrixBlock library instruments all the basic operations on matrix blocks, with different implementations for the same operator based on the characteristics of the input matrix blocks. For example, there are multiple implementations of matrix block multiplication for when both matrices are dense, both are sparse, the first is sparse while the second is dense, and the first is dense whereas the second is sparse. In addition, lightweight database compression techniques can be applied to matrix blocks, and then linear algebra operations, such as matrix-vector multiplication, can be executed directly on the compressed representations [5].

Hybrid Runtime SystemML generates hybrid runtime execution plans that range from in-memory single node execution (CP) to large-scale cluster execution of operators on either MR or Spark [3], hence enabling scaling up or down of computation. For cluster execution, SystemML utilizes resource negotiation frameworks like YARN [14], to achieve automatic resource elasticity in a cluster [8]. Besides data parallelism that would be normally achieved using MR or Spark, SystemML also supports task parallelism through which independent iterations in loops can be executed in parallel [4]. Furthermore, SystemML also strives to achieve the numerical accuracy [13] of statistical operations in addition to efficiency.

7.5 Comparison of the Matrix-Based Systems

In a high level, the three matrix-based graph systems share commonalities, such as exposing a matrix-based interface for graph analytics and relying on a underlying general-purpose distributed processing system for execution. But they are also very different in many aspects. Table 7.1 summaries the major differences of the three systems.

In terms of supported analytics, PEGASUS and GBASE are purely designed for graph analytics, whereas SystemML is for general machine learning (including graph analytics). As a result, SystemML supports general linear algebra as its primitive operations, beyond just matrix-vector multiplication. Although all are matrix based, the three systems provide very different interfaces, hence different levels of flexibilities to the users: GBASE only exposes built-in algorithms for users to choose, PEGASUS allows users to customize the API implementation for matrix-vector multiplications, whereas SystemML provides a rich scripting language to compose a graph/machine learning algorithm using linear algebra operations.

Table 7.1 Comparison of PEGASUS, GBASE and SystemML

	PEGASUS	GBASE	SystemML
Supported analytics	Global graph analysis	Targeted and global graph analysis	Graph analysis and machine learning
Core operations	Matrix-vector multiplication	Matrix-vector multiplication	General linear algebra
User interface	Customizable APIs	Built-in algorithms	R or Python-like language
Matrix blocking	Square	General rectangular	General rectangular
Block formation	Clustering nodes	Clustering nodes	No clustering needed
Matrix storage	Uncompressed	Compressed, special placement	Compressed, per-block layout
Query optimization	NA	NA	Cost-based and rule-based
Runtime platform	MapReduce	MapReduce	Single node and MapReduce/Spark

All three systems break a matrix into blocks for efficient processing and storage. PEGASUS only uses square blocks, whereas the other two support general rectangular blocks. Both PEGASUS and GBASE require a preprocessing step to cluster the nodes in order to generate compact blocks. In terms of storage, both GBASE and SystemML support compression on the blocks. To support efficient access to a matrix, GBASE uses a grid placement to minimize file access, whereas SystemML optimizes the physical layout at the matrix-block level.

Although a number of system optimizations are used in PEGASUS and GBASE, these two systems do not apply any query-specific optimizations. Taking a declarative approach, SystemML employs both cost-based and rule-based optimizations for each graph/machine learning algorithm. In addition, SystemML also generates a hybrid runtime to scale up and down the computation.

7.6 Matrix-Based vs. Vertex-Centric Systems

Finally, let's compare the matrix-based graph systems to the vertex-centric graph systems (which loosely include the block-centric graph systems in this discussion).

Which interface is better, matrix-based or vertex-centric, is in the eye of the beholder. For data analysts who are more familiar with writing algorithms using linear algebra operations in R or MATLAB, the matrix-based systems may be more intuitive to them. But for programmers who are comfortable writing code, the vertex-centric systems can give them more flexibility and control, as they can express more customized logic for their graph algorithms, which can be hard to express using matrix operations.

As for the runtime, the three matrix-based systems all rely on an existing general-purpose data processing engine for execution, whereas many vertex-centric graph systems implement their own runtime engines specially designed for graph processing. On one hand, this makes graph analytics on the matrix-based systems more easily integrated with other types of data processing tasks, such as Extract-Transform-Load (ETL) and machine learning, on the same underlying processing platform. On the other hand, this also means that the vertex-centric systems can often avoid some unnecessary overhead incurred in matrix-based systems. For example, MapReduce materializes results after each Map-and-Reduce phase. As a result, the matrix-based systems built on MapReduce have to read and write graph data many times throughout an algorithm. In comparison, many vertex-centric systems can keep graph data in memory across iterations without incurring IO.

In addition, vertex-centric systems often keep track of whether a vertex is active or not throughout an algorithm, so that computation is only occurred on active vertices. Many iterative graph algorithms, such PageRank and Connected Component, only involve a very small number of active vertices for computation, as the algorithm approaches to its convergence. But in matrix-based systems, each iteration requires a full matrix computation, even though only a few of the matrix elements will change their values. As a result, the matrix-based systems may show some disadvantage in performance compared to the vertex-centric systems.

In summary, both matrix-based and vertex-centric graph systems have their pros and cons. There are many factors that affect the choice of a system for a particular user, including the intuitiveness and the expressiveness of the user interface, the ease of integration with other analytics tasks, and of course the runtime performance.

References

1. N. Biggs. *Algebraic Graph Theory*. Cambridge University Press, 2nd edition, 1993.
2. M. Boehm, D. Burdick, A. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. SystemML's optimizer: Plan generation for large-scale machine learning programs. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 37(3), 2014.
3. M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda. SystemML: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.
4. M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *PVLDB*, 7(7):553–564, 2014.
5. A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12):960–971, 2016.
6. A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.
7. C. Godsil and G. Royle. *Algebraic Graph Theory*, volume 207 of Graduate Texts in Mathematics. Springer, 2001.
8. B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.
9. U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos. GBASE: a scalable and general graph management system. In *SIGKDD*, pages 1091–1099, 2011.
10. U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. gbase: an efficient analysis platform for large graphs. *The VLDB Journal*, 21(5):637–650, 2012.
11. U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.
12. U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: Mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, May 2011.
13. Y. Tian, S. Tatikonda, and B. Reinwald. Scalable and numerically stable descriptive statistics in SystemML. In *ICDE*, pages 1351–1359, 2012.
14. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, pages 5:1–5:16, 2013.

Chapter 8
Conclusions

This chapter concludes the book with a summary, and provides some vision for future research directions.

8.1 Summary

Chapter 2 introduces the vertex-centric computation model pioneered by Google, and the other Pregel-like systems with various optimization techniques. The topic is presented at the very beginning since Pregel-like computation model is unarguably the most popular model for processing big graphs, and many concepts involved are important for understanding the other graph analytics frameworks. Chapter 3 then presents a tutorial on the BigGraph@CUHK toolkit, and the Pregel+ system in specific. The tutorial not only provides a timely hands-on experience of using a Pregel-like system, but also includes numerous valuable information on how to design a Big Data system from scratch. As the last chapter on vertex-centric computation, Chap. 4 reviews those systems that adopt a different, shared-memory programming abstraction. While GraphLab is a representative system in this category, we pointed out the weaknesses in its design such as high data replication rate, and mentions other alternative (e.g., Maiter) that could provide the same benefit of GraphLab (i.e., faster convergence) with a higher efficiency.

Chapter 5 introduces the block-centric computation model that overcomes the performance weaknesses of vertex-centric model in processing real graphs with properties unfavorable to vertex-centric computation. The chapter also introduces the state-of-the-art block-centric system, Blogel, in detail. Chapter 6 introduces the subgraph-centric computation model for mining a big graph, which is not suitable for vertex-centric computation. The chapter also introduces the state-of-the-art subgraph-centric system, G-thinker, in detail.

Finally, Chap. 7 introduces matrix-based systems for big graph analytics, with a special emphasis on SystemML. SystemML target machine learning applications (including graph analytics), and translates high-level Declarative Machine learning Language (DML) for optimized execution with Spark or MapReduce.

8.2 Future Research Directions

The vertex-centric model for processing big graphs is still a young research direction, given that Pregel was only proposed in 2010 followed by GraphLab and GraphChi in 2012. Although many graph systems have been proposed in recent years, there are still many aspects that have not been explored. It is also necessary to design a richer set of vertex-centric algorithms for various graph analytics tasks, as currently the number of system papers in this field far outstrips the number of algorithm papers.

We notice that vertex-centric systems are restricted in expressiveness, and is typically for solving graph problems with low complexity. For example, PPA presented in Sect. 2.1.2 requires each iteration to have $O(|G|)$ cost where $|G|$ is the size of the input graph, and the number of iterations to be $O(\log |G|)$; in other words, the cost of a PPA is about $O(|G| \log |G|)$, much lighter than many graph problems such as graph matching and subgraph pattern mining. It is interesting to look into computation models that can handle computation-intensive graph analytics tasks, such as the subgraph-centric framework presented in Chap. 6. We expect this direction to be promising, since existing Big Data frameworks including MapReduce, Spark and Pregel are all designed for data-intensive workloads, and the research for computation-intensive Big Data analytics is inadequate.

It is also interesting to look into system architectures that support efficient big matrix/tensor analytics. Existing solutions to big matrix processing include (1) to design a system with native support for big array storage and processing, such as SciDB[1] and (2) to build a layer on top of MapReduce or Spark, such as SystemML[2] and SparkR.[3] Both solutions support only simple matrix operations such as obtaining a submatrix, or to perform matrix multiplication/addition, while the latter further relies on a data-intensive framework for execution. Many high-complexity matrix operations such as matrix decompositions are not well supported, which leaves much room for improvement.

[1] http://www.paradigm4.com/.

[2] https://systemml.apache.org/.

[3] http://spark.apache.org/docs/latest/sparkr.html.