Distributed Graph Computing Systems: Design, Implementation and Applications

Lu, Yi

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
September 2015

Last updated: January 6, 2015

Abstract of thesis entitled:
Distributed Graph Computing Systems: Design, Implementation and Applications
Submitted by Lu, Yi

With the growing interest in analyzing large real-world graphs such as online social networks, mobile communication networks and semantic web graphs, many distributed graph computing systems have emerged. These systems are deployed in a shared-nothing distributed computing infrastructure usually built on top of a cluster of low-cost commodity PCs. Pioneered by Google's Pregel, these systems adopt a vertex-centric computing paradigm, where programmers think naturally like a vertex when designing distributed graph algorithms.

In this thesis, we first investigate the limitations of Pregel's simple message passing mechanism, with which some vertices may send/receive significantly more messages than others due to either the high degree of these vertices or the logic of an algorithm being implemented. As a solution to this problem, we propose two effective message reduction techniques, which not only reduce the total number of messages exchanged through the network, but also bound the number of messages sent/received by any single vertex. We theoretically analyze the effectiveness of our techniques, and implement them on top of our open-source Pregel implementation called Pregel+.

Next, we describe the design and implementation of Blogel, a new graph computing system with a "think like a block" programming model. Blogel programmers may think like a block and develop efficient algorithms for various graph problems. We propose parallel algorithms to partition an arbitrary graph into blocks efficiently, and block centric programs (e.g. PageRank, Weakly Connected Components, Single Source Shortest Path and Reachability) are then run over these blocks.

Then, we identify a set of desirable properties of an efficient Pregel algorithm, such as linear space, communication and computation cost per iteration, and logarithmic number of iterations. We define such an algorithm as a practical Pregel algorithm (PPA) and propose PPAs for graph connectivity problems. We then conduct extensive experiments to evaluate the performance of existing systems on graphs with different characteristics and on algorithms with different design logic. We also study the effectiveness of various techniques adopted in existing systems, and the scalability of the systems. The results of our study reveal the strengths and limitations of existing systems, and provide valuable insights for users, researchers and system developers.

Finally, we remark that our systems, Pregel+ and Blogel, have been deployed and used in Taobao Data Lab, for analyzing massive amounts of graph data from Taobao's online shopping network.

隨著人們對線上社交網，移動通信網絡，與語意網越來越多的關注，許多分佈式圖計算系統應運而生。這些系統大多採用了由Google提出的以頂點爲中心的程式設計范式，且通常部署在無共用資源結構的低成本PC上。

在本論文中，我們首先觀察到在頂點度數分佈不均勻的圖上或者某些算法中，一些頂點由於Pregel單一的消息傳遞機制會比其它頂點接收或發送更多消息的現象。爲了解決這一問題，我們提出了兩種解決方案，使其不僅能減少網路中傳遞的消息總數，也能限制圖中任意頂點發送或接受消息的條數。我們從理論上分析了這些技術的有效性，並在開源系統Pregel+加以了實現。

接下來，我們探討了以圖塊爲中心的Blogel的設計與實現。Blogel的開發者可以通過圖塊爲基礎來開發高效的圖算法。同時，我們還提出了高效的分佈式圖劃分算法和以這些圖塊爲基礎的圖算法（PageRank，弱連通分支，單源最短路及可達性問題）。

之後，我們總結了一套高效Pregel算法應該具有的特性，比如每輪運算只需要線性空間，線性通信與計算代價。我們將這種算法定義爲實用的Pregel算法（PPA），並且提出瞭解決圖連通性問題的Pregel 算法（PPA）。之後我們又利用各種不同的算法和圖對現有系統進行了性能的評估，同時也分析了各種系統層面和算法層面優化技巧的有效性，及各系統的可擴充性。我們的研究成果可以揭示現有系統的長處與局限性，並爲使用者，研究員及系統開發員提供寶貴的見解。

最後，我們所開發的Pregel+和Blogel系統已經部署在淘寶數據實驗室，用以分析淘寶在線購物的圖數據。

Thesis Assessment Committee

Professor TAO Yufei (Chair)
Professor CHENG James (Thesis Supervisor)
Professor LUI, John C.S. (Committee Member)

Contents

1 Introduction **1**

1.1 Pregel+ Architecture 1

1.2 Blogel Architecture 6

1.3 Pregel Algorithms with Performance Guarantee 8

1.4 Extensive Performance Study 9

2 Pregel+ Architecture **11**

2.1 Background and Related Work 12

2.2 Pregel Algorithms 14

2.3 Overview of the Basic Pregel+ System 18

2.4 The Mirroring Technique 22

2.5 The Request-Respond Paradigm 27

2.6 Experimental Results 30

2.7 Conclusions 35

3 Blogel Architecture **36**

3.1 Background and Related Work 37

3.2 Merits of Block-Centric Computing: a First Example 39

3.3 System Overview 42

3.4 Programming Interface 45

3.5 Applications 48

3.6 Partitioners 52

3.7 Experiments 55

3.8 Conclusions 64

4 Pregel Algorithms with Performance Guarantee **65**

4.1 Related Work 65

4.2 Practical Pregel Algorithms 67

4.3 Connected Components 69
4.4 Bi-Connected Components 74
4.5 Strongly Connected Components 83
4.6 Experimental Evaluation 91
4.7 Conclusions 102

5 Extensive Performance Study 103
5.1 Preliminary 105
5.2 A Survey on Existing Systems 105
5.3 Algorithms 110
5.4 Experimental Evaluation 115
5.5 Conclusions 129

6 Concluding Remarks 131

List of Figures

1.1 Hash-Min on BTC (with/without mirroring) 4
1.2 S-V on USA (with/without request-respond) 5
1.3 Illustration of Req-Resp API 10
1.4 Systems overview 10

2.1 Illustration of combiner 12
2.2 Forest structure of the S-V algorithm 16
2.3 Key operations of the S-V algorithm 17
2.4 Conjoined Tree 17
2.5 Illustration of Message Channel, Ch_{msg} 20
2.6 Message # of Each Worker: Hash-Min on BTC in GPS 21
2.7 Illustration of Mirroring 23
2.8 Mirroring v.s. Message Combining 24
2.9 Illustration of request-respond paradigm 28
2.10 Datasets (M=1,000,000) 31
2.11 Effects of mirroring ($\star$: best result, M = million) 32
2.12 Effects of message combiner 33
2.13 Effects of the Request-Respond Paradigm 34

3.1 Overall Performance of Hash-Min 40
3.2 Performance of Hash-Min Per Superstep 40
3.3 Operating Logic of Different Types of Workers 43
3.4 Programming Interface of Blogel 46
3.5 Impact of PageRank Loss 51
3.6 Partitioners (Best Viewed in Colors) 53
3.7 Datasets 56
3.8 Performance of Graph Voronoi Diagram Partitioners 57
3.9 # of Blocks/Vertices Per Worker (GVD Partitioner) 59
3.10 Performance of 2D Partitioners 59

3.11 Partitioning Performance of Giraph++ 60
3.12 Partitioning Performance of LDG 60
3.13 GVD Partitioner Scalability on Real Graphs 61
3.14 GVD Partitioner Scalability on Random Graphs 61
3.15 Performance of CC, SSSP and Reachability Computation 62
3.16 Performance of PageRank Computation 63

4.1 Forest structure of Shiloach-Vishkin's algorithm 70
4.2 Tree hooking, star hooking, and shortcutting 71
4.3 Illustration of the change to star hooking 71
4.4 BCC illustration 74
4.5 Illustration of construction of G^* 76
4.6 Euler tour 77
4.7 Illustration of BPPA for list ranking 79
4.8 Pre-order & post-order 80
4.9 Illustration of computing $min(v)$ 83
4.10 Concepts in SCC computation 84
4.11 Datasets 92
4.12 Pregel+ and GraphLab running Hash-Min 93
4.13 CC/BCC performance on BTC, LJ-UG, & Facebook 95
4.14 CC/BCC performance on USA & Euro 96
4.15 Min-label performance on Twitter ($\tau = 0$) 97
4.16 Min-label performance on LJ-DG ($\tau = 0$) 97
4.17 Min-label performance on Pokec ($\tau = 0$) 98
4.18 Min-label performance on Flickr ($\tau = 0$) 98
4.19 Min-label performance on Patent ($\tau = 0$) 99
4.20 Multi-label performance on Twitter ($\tau = 50,000$) 99
4.21 Multi-label performance on LJ-DG ($\tau = 50,000$) 100
4.22 Multi-label performance on Pokec ($\tau = 50,000$) 101
4.23 Multi-label performance on Flickr ($\tau = 50,000$) 101
4.24 Multi-label performance on Patent ($\tau = 50,000$) 101

5.1 Datasets 115
5.2 Performance overview on Giraph, GPS, GraphLab, and Pregel+ 116
5.3 Performance of asynchronous computing (in GraphLab) and synchronous computing 117
5.4 Asynchronous PageRank in GraphLab 118
5.5 Overall performance on different algorithms 119
5.6 Overall performance on different graphs 122

5.7 Performance of GraphChi v.s. Pregel+ on PageRank, Diameter Est., HashMin, SSSP and BMM . . . 122
5.8 Effects of combiner in Giraph and Pregel+ with different number of workers 122
5.9 Diameter Est on LJ (LALP and mirroring) . . . 124
5.10 HashMin on BTC (LALP and mirroring) . . . 125
5.11 Effects of request-respond API in Pregel+ . . . 125
5.12 Effects of dynamic repartitioning . . . 125
5.13 Effects of ECOD, FCS and SP . . . 126
5.14 Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of machines or CPU cores . . . 127
5.15 Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of vertices . . . 127
5.16 Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of edges . . . 128

Chapter 1

Introduction

Many distributed graph computing systems have emerged to conduct efficient and effective analysis on massive real-world graphs, such as online social networks, mobile communication networks and semantic web graphs. Most of these systems are built upon a shared-nothing distributed computing infrastructure and adopt the vertex-centric model proposed in [33], including Pregel+ [55], Blogel [54], GraphLab [30], PowerGraph [17], GPS [40], Giraph [1], Giraph++ [49].

Although Pregel's vertex-centric computing model has been widely adopted, the vertex-centric model has largely ignored the characteristics of real-world graphs and algorithms in its design and can hence suffer from severe performance problems. To address the performance bottlenecks in vertex-centric systems, we propose Pregel+ and Blogel. Pregel+ is beyond yet-another-Pregel-like system, but it integrates two effective message reduction techniques to significantly improve the performance of existing Pregel-like systems. Blogel adopts a new programming model, block-centric computing model, and naturally addresses a number of performance limitations of Pregel in handling real-world graphs.

We also identify a set of rigid, but practical constraints on various performance metrics for algorithm design to avoid ad hoc algorithm design, thus ensuring good performance. Under such constraints, we study three fundamental graph connectivity problems and propose Pregel algorithms as solutions. Finally, we use different categories of algorithms to study the performance of existing systems on graphs with different characteristics and on algorithms with different design logic.

1.1 Pregel+ Architecture

With the growing interest in analyzing large real-world graphs such as online social networks and semantic web graphs, many distributed graph computing systems have emerged.

These systems are deployed in a shared-nothing distributed computing infrastructure usually built on top of a cluster of low-cost commodity PCs. Pioneered by Google's Pregel [33], these systems adopt a vertex-centric computing paradigm, where programmers think naturally like a vertex when designing distributed graph algorithms. A Pregel-like system also takes care of fault recovery and scales to arbitrary cluster size without the need of changing the program code, both of which are indispensable properties for programs running in a cloud environment.

MapReduce [11], and its open-source implementation Hadoop[1], are another distributed system popularly used for large scale graph processing [22, 28, 23, 46, 37]. However, many graph algorithms are intrinsically iterative, such as the computation of PageRank or shortest paths. For iterative graph computation, a Pregel program is much more efficient than its MapReduce counterpart. This is because each MapReduce job can only perform one iteration of graph computation, and it requires to read the entire graph from a distributed file system (DFS) and write the processed graph back, which leads to excessive IO cost. Furthermore, a MapReduce job also needs to exchange the adjacency lists of vertices through the network, which results in heavy communication. In contrast, Pregel keeps vertices (along with their adjacency lists) in each local machine that processes their computation, and uses only message passing to exchange vertex states. Pregel's "think like a vertex" programming paradigm is also easier to use than MapReduce when designing graph algorithms.

Weaknesses of Pregel. Although Pregel's vertex-centric computing model has been widely adopted in most of the recent distributed graph computing systems [1, 30, 25, 40] (and also inspired the edge-centric system [17]), Pregel's vertex-to-vertex message passing mechanism often causes bottlenecks in communication when processing real-world graphs.

To clarify this point, we first briefly review how Pregel performs message passing. In Pregel, a vertex v can send messages to another vertex u if v knows u's vertex ID. In most cases, v only sends messages to its neighbors whose IDs are available from v's adjacency list. But there also exist Pregel algorithms in which a vertex v may send messages to another vertex that is not a neighbor of v [56, 41]. These algorithms usually adopt pointer jumping (or doubling), a technique that is widely used in designing PRAM algorithms [44], to bound the number of iterations by $O(\log |V|)$, where $|V|$ refers to the number of vertices in the graph.

The problem with Pregel's message passing mechanism is that a small number of vertices, which we call *bottleneck vertices*, may send/receive much more messages than other vertices. A bottleneck vertex not only generates heavy communication, but also significantly increases the workload of the machine in which the vertex resides, causing highly

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

imbalanced workload among different machines. Bottleneck vertices are common when using Pregel to process real-world graphs, mainly due to either (1)high vertex degree or (2)algorithm logic, which we elaborate more as follows.

We first consider the problem caused by high vertex degree. When a high-degree vertex sends messages to all its neighbors, it becomes a bottleneck vertex. Unfortunately, real-world graphs usually have highly skewed degree distribution, with some vertices having very high degrees. For example, in the *Twitter* who-follows-who graph[2], the maximum degree is over 2.99M while the average degree is only 35. Similarly, in the *BTC* dataset used in our experiments, the maximum degree is over 1.6M while the average degree is only 4.69.

We ran *Hash-Min* [37, 56], a distributed algorithm for computing connected components (CCs), on the degree-skewed *BTC* dataset in a cluster with 1 master (Worker 0) and 120 slaves (Workers 1–120), and observed highly imbalanced workload among different workers, which we describe next. Pregel assigns each vertex to a worker by hashing the vertex ID regardless of the degree the vertex. As a result, each worker holds approximately the same number of vertices, but the total number of items in the adjacency lists (i.e., edges) varies greatly among different workers. In the computation of *Hash-Min* on *BTC*, we observed an uneven distribution of edge number among workers, as some workers contain more high-degree vertices than other workers. Since messages are sent along the edges, the uneven distribution of edge number also leads to an uneven distribution of the amount of communication among different workers. In Figure 1.1, the taller blue bars indicate the total number of messages sent by each worker during the entire computation of *Hash-Min*, where we observe highly uneven communication workload among different workers.

Bottleneck vertices may also be generated by program logic. An example is the *S-V* algorithm proposed in [56, 44] for computing CCs, which we will describe in detail in Section 2.2.4. In *S-V*, each vertex v maintains a field $D[v]$ which records the vertex that v is to communicate with. The field $D[v]$ may be updated at each iteration as the algorithm proceeds, and when the algorithm terminates, all vertices v in the same CC have the same value of $D[v]$. Thus, during the computation, a vertex u may communicate with many vertices $\{v_1, v_2, \ldots, v_\ell\}$ in its CC if $u = D[v_i]$ for $1 \leq i \leq \ell$. In this case, u becomes a bottleneck vertex.

We ran *S-V* on the USA road network in a cluster with 1 master (Worker 0) and 60 slaves (Workers 1–60), and observed highly imbalanced communication workload among different workers. In Figure 1.2, the taller blue bars indicate the total number of messages sent by each worker during the entire computation of *S-V*, where we can see that the communication workload is very biased (esp. at Worker 0). We remark that the imbalanced

[2] http://law.di.unimi.it/webdata/twitter-2010/

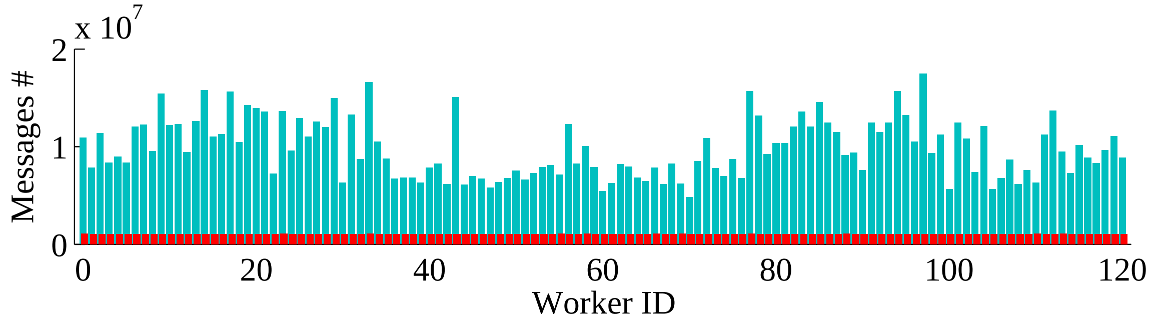


Figure 1.1: Hash-Min on BTC (with/without mirroring)

communication workload is not caused by skewed vertex degree distribution, since the largest vertex degree of the USA road network is merely 9. Rather, it is because of the logic of *S-V*. Specifically, since the USA road network is connected, in the last round of *S-V*, all vertices v have $D[v]$ equal to Vertex 0, indicating that they all belong to the same CC. Since Vertex 0 is hashed to Worker 0, Worker 0 sends much more messages than the other workers, as can be observed from Figure 1.2.

In addition to the two problems mentioned above, Pregel's message passing mechanism is also not efficient for processing graphs with (relatively) high average degree due to the high overall communication cost. However, graphs like social networks and mobile phone networks have relatively high average degree, as a person is often connected to at least dozens of people.

Our Solution. In this thesis, we solve the problems caused by Pregel's message passing mechanism with two effective message reduction techniques. The goals are to *(1)mitigate the problem of imbalanced workload by eliminating bottleneck vertices*, and to *(2)reduce the overall number of messages exchanged through the network*.

The first technique is called **mirroring**, which is designed to eliminate bottleneck vertices caused by high vertex degree. The main idea is to construct mirrors of each high-degree vertex in different machines, so that messages from a high-degree vertex are forwarded to its neighbors by its mirrors in local machines. Let $d(v)$ be the degree of a vertex v and M be the number of machines in the cluster, mirroring bounds the number of messages sent by v each time to $\min\{M, d(v)\}$. If v is a high-degree vertex, $d(v)$ can be up to millions, but M is normally only from tens to a few hundred. However, as we shall see in Section 2.4, mirroring a vertex does not always reduce the number of messages due to Pregel's use of message combiner [33]. Hence, we provide a theoretical analysis on which vertices should be selected for mirroring.

In Figure 1.1, the short red bars indicate the total number of messages sent by each

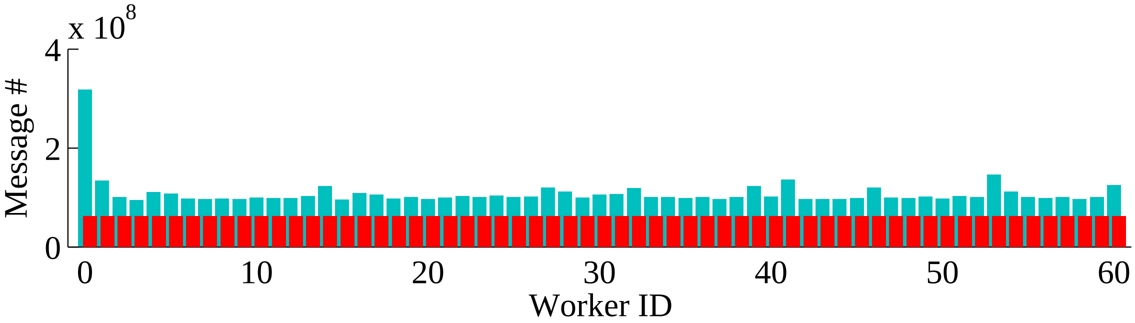


Figure 1.2: S-V on USA (with/without request-respond)

worker when mirroring is applied to all vertices with degree at least 100. We can clearly see the big difference between the uneven blue bars (without mirroring) and the even-height short red bars (with mirroring). Furthermore, the number of messages is also significantly reduced by mirroring. We remark that the algorithm is still the same and mirroring is completely transparent to users. Mirroring reduces the running time of *Hash-Min* on BTC from 26.97 seconds to 9.55 seconds.

The second technique is a new **request-respond paradigm**. We extend the basic Pregel framework by an additional request-respond functionality. A vertex u may request another vertex v for its attribute $a(v)$, and the requested value will be available in the next iteration. The request-respond programming paradigm simplifies the coding of many Pregel algorithms, as otherwise at least three iterations are required to explicitly code each request and response process. More importantly, the request-respond paradigm effectively eliminates the bottleneck vertices resulted from algorithm logic, by bounding the number of response messages sent by any vertex to M. Consider the S-V algorithm mentioned earlier, where vertices $\{v_1, v_2, \ldots, v_\ell\}$ with $D[v_i] = u$ require the value of $D[u]$ from u (thus there are ℓ requests and responses). Under the request-respond paradigm, all the requests from a machine to the same target vertex are merged into one request. Therefore, at most $\min\{M, \ell\}$ requests are needed for the ℓ vertices and at most $\min\{M, \ell\}$ responses are sent from u. For large real-world graphs, ℓ is often orders of magnitude greater than M.

In Figure 1.2, the short red bars indicate the total number of messages sent by each worker when the request-respond paradigm is applied. Again, the skewed message passing represented by the blue bars are now replaced by the even-height short red bars. In particular, Vertex 0 now only responds to the requesting workers instead of all the requesting vertices in the last round, and hence the highly imbalanced workload caused by Vertex 0 in Worker 0 is now evened out. The request-respond paradigm reduces the running time of *S-V* on the USA road network from 261.9 seconds to 137.7 seconds.

Finally, we remark that our experiments were run in a cluster without any resource contention, and our optimization techniques are expected to improve the overall performance of Pregel algorithms more significantly if they were run in a public data center, where the network bandwidth is lower and reducing communication overhead becomes more important.

1.2 Blogel Architecture

Due to the growing need to deal with massive graphs in various graph analytic and graph mining applications, many distributed graph computing systems have emerged in recent years, including Pregel [33], GraphLab [30], PowerGraph [17], Giraph [1], GPS [40], and Mizan [25]. Most of these systems adopt the *vertex-centric* model proposed in [33], which promotes the philosophy of "thinking like a vertex" that makes the design of distributed graph algorithms more natural and easier. However, the vertex-centric model has largely ignored the characteristics of real-world graphs in its design and can hence suffer from severe performance problems.

We investigated a broad spectrum of real-world graphs and identified three characteristics of large real-world graphs, namely (1)*skewed degree distribution* (common for power-law and scale free graphs such as social networks and web graphs), (2)*(relatively) high density* (common for social networks, mobile phone networks, SMS networks, some web graphs, and the cores of most large graphs), and (3)*large diameter* (common for road networks, terrain graphs, and some large web graphs). These three characteristics are particularly adverse to vertex-centric parallelization as they are often the major cause(s) to one or more of the following three performance bottlenecks: *skewed workload distribution*, *heavy message passing*, and *impractically many rounds of computation*.

Let us first examine the performance bottleneck created by skewed degree distribution. The vertex-centric model assigns each vertex together with its adjacency list to a machine, but neglects the difference in the number of neighbors among different vertices. As a result, for graphs with skewed degree distribution, it creates unbalanced workload distribution that leads to a long elapsed running time due to waiting for the last worker to complete its job. For example, the maximum degree of the BTC RDF graph used in our experiments is 1,637,619, and thus a machine holding such a high-degree vertex needs to process many incoming messages and send out many outgoing messages to its neighbors, causing imbalanced workload among different machines.

Some existing systems proposed techniques for better load balancing [17, 40], but they do not reduce the overall workload. However, for many real-world graphs including power-law graphs such as social networks and mobile phone networks, the average vertex degree is large. Also, most large real-world graphs have a high-density core (e.g., the k-core [43] and k-truss [50] of these graphs). Higher density implies heavier message passing for vertex-centric systems. We show that heavy communication workload due to high density can also be eliminated by our new computing model.

For processing graphs with a large diameter δ, the message (or neighbor) propagation paradigm of the vertex-centric model often leads to algorithms that require $O(\delta)$ rounds (also called supersteps) of computation. For example, a single-source shortest path algorithm in [33] takes 10,789 supersteps on a USA road network. Apart from spatial networks, some large web graphs also have large diameters (from a few hundred to thousands). For example, the vertex-centric system in [39] takes 2,450 rounds for computing strongly connected components on a web graph.

To address the performance bottlenecks created by real-world graphs in vertex-centric systems, we propose a *block-centric* graph-parallel abstraction, called **Blogel**. Blogel is conceptually as simple as Pregel but works in coarser-grained graph units called *blocks*. Here, a block refers to a connected subgraph of the graph, and message exchanges occur among blocks.

Blogel naturally addresses the problem of skewed degree distribution since most (or all) of the neighbors of a high-degree vertex v are inside v's block, and they are processed by sequential in-memory algorithms without any message passing. Blogel also solves the heavy communication problem caused by high density, since the neighbors of many vertices are now within the same block, and hence they do not need to send/receive messages to/from each other. Finally, Blogel effectively handles large-diameter graphs, since messages now propagate in the much larger unit of blocks instead of single vertices, and thus the number of rounds is significantly reduced. Also, since the number of blocks is usually orders of magnitude smaller than the number of vertices, the workload of a worker is significantly less than that of a vertex-centric algorithm.

A central issue to Blogel is whether we can partition an arbitrary input graph into blocks efficiently. We propose a *graph Voronoi diagram based partitioner* which is a fully distributed algorithm, while we also develop more effective partitioners for graphs with additional information available. Our experiments show that our partitioning algorithms are efficient and effective.

We present a user-friendly and flexible programming interface for Blogel, and illustrate that programming in Blogel is easy by designing algorithms for a number of classic graph problems in the Blogel framework. Our experiments on large real-world graphs with up to hundreds of millions of vertices and billions of edges, and with different characteristics, verify that our block-centric system is orders of magnitude faster than the state-of-the-art vertex-centric systems [1, 30, 17]. We also demonstrate that Blogel can effectively address the performance bottlenecks caused by the three adverse characteristics of real-world graphs.

1.3 Pregel Algorithms with Performance Guarantee

The popularity of online social networks, mobile communication networks and semantic web services, has stimulated a growing interest in conducting efficient and effective analysis on massive real-world graphs. To process such large graphs, Google's Pregel [33] proposed the vertex-centric computing paradigm, which allows programmers to think naturally like a vertex when designing distributed graph algorithms. A Pregel-like system runs on a shared-nothing distributed computing infrastructure which can be deployed easily on a cluster of low-cost commodity PCs. The system also removes from programmers the burden of handling fault recovery, which is important for programs running in a cloud environment.

Pregel was shown to be more suitable for iterative graph computation than the popular MapReduce model [10, 33, 36]. However, existing work on Pregel algorithms [36] is rather ad hoc, which looks more like a demonstration of how Pregel can be used to solve a number of graph problems, and lacks any analysis on the cost complexity.

In this thesis, we study three fundamental graph connectivity problems and propose Pregel algorithms as solutions that have performance guarantees. The problems we study are *connected components* (**CC**s), *bi-connected components* (**BCC**s), and *strongly connected components* (**SCC**s). These problems have numerous real life applications and their solutions are essential building blocks for solving many other graph problems. For example, computing the BCCs of a telecommunications network can help detect the weaknesses in network design, while almost all reachability indices require SCC computation as a preprocessing step [8].

To avoid ad hoc algorithm design and ensure good performance, we define a class of Pregel algorithms that satisfy a set of rigid, but practical, constraints on various performance metrics: *(1)linear space usage, (2)linear computation cost per iteration*[1], *(3)linear communication cost per iteration, and (4)at most logarithmic number of iterations*. We call such algorithms as **Practical Pregel Algorithm**s (**PPA**s). A similar but stricter set of constraints was proposed for the MapReduce model recently [47]. In contrast to our requirement of logarithmic number of iterations, their work demands a constant number of iterations, which is too restrictive for most graph problems. In fact, even list ranking (i.e., ranking vertices in a directed graph consisting of only one simple path) requires $O(\log n)$ time using $O(n)$ processors under the *shared-memory* PRAM model [51], where n is the

[1] A Pregel algorithm proceeds in iterations.

number of vertices. This bound also applies to many other basic graph problems such as connected components and spanning tree [44].

It is challenging to design Pregel algorithms for problems such as BCCs and SCCs. Although there are simple sequential algorithms for computing BCCs and SCCs based on depth-first search (DFS), DFS is $\mathcal{P}$-Complete [38] and hence it cannot be applied to design parallel algorithms for computing BCCs and SCCs.

We apply the principle of PPA to develop Pregel algorithms that satisfy strict performance guarantees. In particular, to compute BCCs and SCCs, we develop a set of useful building blocks that are the PPAs of fundamental graph problems such as *breadth-first search*, *list ranking*, *spanning tree*, *Euler tour*, and *pre/post-order traversal*. As fundamental graph problems, their PPA solutions can also be applied to numerous other graph problems besides BCCs and SCCs considered in this thesis.

We evaluate the performance of our Pregel algorithms using large real-world graphs with up to hundreds of millions of vertices and billions of edges. Our results verify that our algorithms are efficient for computing CCs, BCCs and SCCs in massive graphs.

1.4 Extensive Performance Study

Many distributed graph computing systems have been proposed to conduct all kinds of data processing and data analytics in massive graphs, including Pregel [33], Giraph [1], GraphLab [30], PowerGraph [16], GraphX [53], Mizan [25], GPS [40], Giraph++ [49], Pregelix [6], Pregel+ [55], and Blogel [54]. These systems are all built on top of a shared-nothing architecture, which makes big data analytics flexible even on a cluster of low-cost commodity PCs.

The majority of the systems adopt a "think like a vertex" vertex-centric computing model [33], where each vertex in a graph receives messages from its incoming neighbors, executes the user-specified computation and updates its value, and then sends messages to its outgoing neighbors. The vertex-centric computing model makes the design and implementation of scalable distributed algorithms simple for ordinary users, while the system handles all the low-level details. There are also a few extensions to the vertex-centric model, e.g., the edge-centric model in PowerGraph [16] and the block-centric models in [49, 54], to address various limitations in the vertex-centric systems. However, these models also suffer from other problems such as longer graph partitioning time.

While many distributed graph computing systems have emerged recently, it is unclear to most users and researchers what the strengths and limitations of each system are and there is a lack of overview on how these systems compare with each other. Thus, it is difficult for users to decide which system is better for their applications, or for researchers

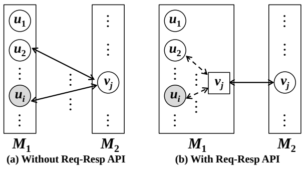
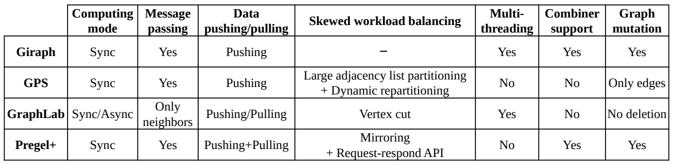


Figure 1.3: Illustration of Req-Resp API



	Computing mode	Message passing	Data pushing/pulling	Skewed workload balancing	Multi-threading	Combiner support	Graph mutation
Giraph	Sync	Yes	Pushing	–	Yes	Yes	Yes
GPS	Sync	Yes	Pushing	Large adjacency list partitioning + Dynamic repartitioning	No	No	Only edges
GraphLab	Sync/Async	Only neighbors	Pushing/Pulling	Vertex cut	Yes	No	No deletion
Pregel+	Sync	Yes	Pushing+Pulling	Mirroring + Request-respond API	No	Yes	Yes

Figure 1.4: Systems overview

and system developers to further improve the performance of existing systems or design new systems.

This thesis focuses on the design, implementation, and applications of distributed graph computing systems. We first propose two systems, Pregel+ [55] and Blogel [54]. Then, we propose a set of performance metrics to guide the design of efficient distributed graph algorithms. Finally, we conduct a comprehensive performance study on different graph computing systems and provide insights on the design and performance of these systems. In the remainder of this chapter, we give the background, motivation and an overview of each of the works mentioned above, which are then detailed in Chapters 2-5.

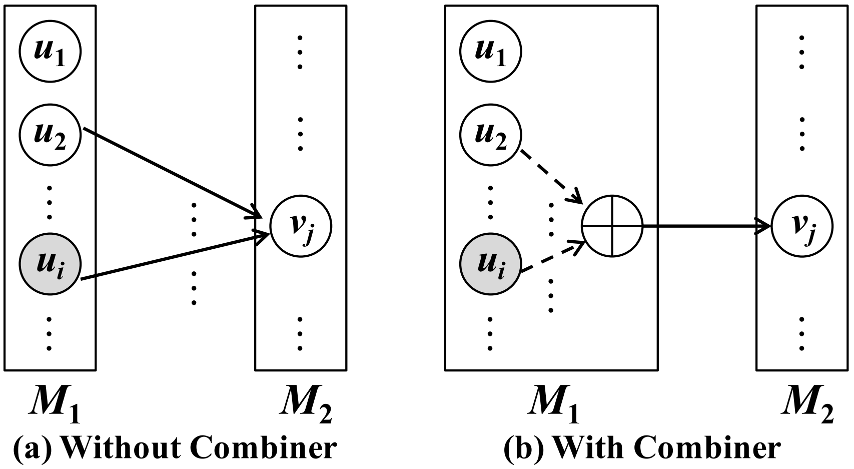
Thesis outline. The thesis will be organized as follows. We present the architecture of Pregel+ with the implementation details of two effective message reduction techniques in Chapter 2. We discuss the novel computing model of Blogel, and propose parallel graph partitioning algorithms and block-centric graph algorithms in Chapter 3 . We identify a set of desirable properties of an efficient Pregel algorithm in Chapter 4. We conduct extensive performance study in Chapter 5. Finally, we give the conclusion of this thesis in Chapter 6.

Chapter 2

Pregel+ Architecture

Pioneered by Google's Pregel, various distributed graph computing systems have been developed in recent years for processing large real-world graphs such as social networks and web graphs. These systems employ the "think like a vertex" programming paradigm, where a program proceeds in iterations and at each iteration, vertices exchange messages with each other. However, using Pregel's simple message passing mechanism, some vertices may send/receive significantly more messages than others due to either the high degree of these vertices or the logic of the algorithm being implemented. This forms the communication bottleneck and leads to imbalanced workload among machines in the cluster. In this chapter, we propose two effective message reduction techniques: (1)vertex mirroring and (2)an additional request-respond API. These techniques not only reduce the total number of messages exchanged through the network, but also bound the number of messages sent/received by any single vertex. We theoretically analyze the effectiveness of our techniques, and implement them on top of our open-source Pregel implementation called Pregel+. Extensive experiments on various large real graphs demonstrate that our message reduction techniques are able to improve the performance of distribution graph computation by up to a few times.

The rest of this chapter is organized as follows. We review existing parallel graph computing systems in Section 2.1, where we discuss the novelty and contribution of this work compared with the existing systems. We describe some Pregel algorithms in Section 2.2, which are later used for system performance evaluation. In Section 2.3, we briefly introduce the design of our Pregel+ system. Our mirroring technique is studied in Section 2.4, and our additional request-respond functionality is described in Section 2.5. Finally, we report the experimental results in Section 2.6 and conclude the chapter in Section 2.7.



(a) Without Combiner (b) With Combiner

Figure 2.1: Illustration of combiner

2.1 Background and Related Work

In this section, we first briefly review the Pregel framework, and then discuss other related distributed graph computing systems.

2.1.1 Pregel

Pregel [33] is designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to different machines in a cluster, where each vertex v is associated with its adjacency list (i.e., the set of v's neighbors). A program in Pregel implements a user-defined *compute*() function and proceeds in iterations (called *supersteps*). In each superstep, the program calls *compute*() for each active vertex. The *compute*() function performs the user-specified task for a vertex v, such as processing v's incoming messages (sent in the previous superstep), sending messages to other vertices (to be received in the next superstep), and making v vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Pregel numbers the supersteps so that a user may use the current superstep number when implementing the algorithm logic in the *compute*() function. As a result, a Pregel algorithm can perform different operations in different supersteps by branching on the current superstep number.

Message Combiner. Pregel allows users to implement a *combine*() function, which specifies how to combine messages that are sent from a machine M_i to the same vertex v in a machine M_j. These messages are combined into a single message, which is then sent from

M_i to v in M_j. However, combiner is applied only when commutative and associative operations are to be applied to the messages. For example, in the PageRank computation, the messages sent to a vertex v are to be summed up to compute v's PageRank value; in this case, we can combine all messages sent from a machine M_i to the same target vertex in a machine M_j into a single message that equals their sum. Figure 2.1 illustrates the idea of combiner, where the messages sent by vertices in machine M_1 to the same target vertex v_j in machine M_2 are combined into their sum before sending.

Aggregator. Pregel also supports aggregator, which is useful 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.

2.1.2 Pregel-Like Systems in JAVA

Since Google's Pregel is proprietary, many open-source Pregel counterparts are developed, such as Giraph [1] and GPS [40]. These systems are implemented in JAVA. They read the graph data from Hadoop's DFS (HDFS) and write the results to HDFS. However, since object deletion is handled by JAVA's Garbage Collector (GC), if a machine maintains a huge amount of vertex/edge objects in main memory, GC needs to track a lot of objects and the overhead can severely degrade the system performance. To decrease the number of objects being maintained, JAVA-based systems maintain vertices in main memory in their binary representation. For example, Giraph organizes vertices as main memory pages, where each page is simply a byte array object that holds the binary representation of many vertices. As a result, a vertex needs to be deserialized from the page holding it before calling *compute*(); and after *compute*() completes, the updated vertex needs to be serialized back to its page. The serialization cost can be expensive, especially if the adjacency list is long. To avoid unnecessary serialization cost, a Pregel-like system needs to be implemented in a language such as C/C++, where programmers (who are system developers, not users) manage main memory objects themselves. Our Pregel+ system is implemented in C/C++, without sacrificing its compatibility with Hadoop as it loads/dumps graph data from/to HDFS using libhdfs (HDFS's C API).

GPS [40] supports an optimization called large adjacency list partitioning (LALP) to handle high-degree vertices, whose idea is similar to vertex mirroring. However, GPS does not explore the performance tradeoff between vertex mirroring and message combining. Instead, [40] claims that very small performance difference can be observed whether combiner is used or not, and thus, GPS simply does not perform sender-side message combining. Our experiments in Section 2.6 show that sender-side message combining significantly reduces the overall running time of Pregel algorithms, and therefore, both vertex mirroring and message combining should be used to achieve better performance. As we shall see in Section 2.4, vertex mirroring and message combining are two conflicting message reduction techniques, and a theoretical analysis on their performance tradeoff is in urgent need in order to guide the choice of vertices for mirroring.

2.1.3 GraphLab and PowerGraph

GraphLab [30] is another parallel graph computing system that follows a different design from Pregel. GraphLab supports asynchronous execution, and adopts a data pulling programming paradigm. Specifically, each vertex actively pulls data from its neighbors, rather than passively receives messages sent/pushed by its neighbors. This feature is somewhat similar to our request-respond paradigm, but in GraphLab, the requests can only be sent to the neighbors. As a result, GraphLab cannot support parallel graph algorithms where a vertex needs to communicates with a non-neighbor. Such algorithms are, however, quite popular in Pregel as they make use of the pointer jumping (or doubling) technique of PRAM algorithms to bound the number of iterations by $O(\log |V|)$. Examples include the S-V algorithm for computing CCs [56] and Pregel algorithm for computing minimum spanning forest [41]. These algorithms are also where our our request-respond paradigm comes into play.

GraphLab also builds mirrors for vertices, which are called ghosts. However, GraphLab creates mirrors for every vertex regardless of its degree, which leads to excessive space consumption. Moreover, GraphLab cannot enjoy the benefit of message combining.

A more recent version of GraphLab, called PowerGraph [17], partitions the graph by edges rather than by vertices. The goal of edge partitioning is to mitigate the problem of imbalanced workload as the edges of a high-degree vertex are handled by multiple workers. As a tradeoff, however, a more complicated edge-centric Gather-Apply-Scatter (GAS) computing model should be used, which compromises user-friendliness.

2.2 Pregel Algorithms

In this section, we describe five Pregel algorithms, which will be used for performance evaluation in our experiments.

We first define the graph notations used in the chapter. Given an undirect graph $G = (V, E)$, we denote the neighbors of a vertex $v \in V$ by $\Gamma(v)$, and the degree of v by $d(v) = |\Gamma(v)|$; if G is directed, we denote the in-neighbors (out-neighbors) of a vertex v by $\Gamma_{in}(v)$ ($\Gamma_{out}(v)$), and the in-degree (out-degree) of v by $d_{in}(v) = |\Gamma_{in}(v)|$ ($d_{out}(v) = |\Gamma_{out}(v)|$). Each vertex $v \in V$ has a unique integer ID, denoted by $id(v)$. The diameter of G is denoted by δ.

2.2.1 Attribute Broadcast

We first introduce a Pregel algorithm for attribute broadcast. Given a directed graph G, where each vertex v is associated with an attribute $a(v)$ and an adjacency list that contains the set of v's out-neighbors $\Gamma_{out}(v)$, *attribute broadcast* constructs a new adjacency list for each vertex v in G, which is defined as $\widehat{\Gamma}_{out}(v) = \{\langle u, a(u)\rangle | u \in \Gamma_{out}(v)\}$.

Put simply, *attribute broadcast* associates each neighbor u in the adjacency list of each vertex v with u's attribute $a(u)$. Attribute broadcast is very useful in distributed graph computation, and it is a frequently performed key operation in many Pregel algorithms. For example, the Pregel algorithm for computing bi-connected components [56] requires to relabel the ID of each vertex u by its preorder number in the spanning tree, denoted by $pre(u)$. Attribute broadcast is used in this case, where $a(u)$ refers to $pre(u)$.

The Pregel algorithm for *attribute broadcast* consists of 3 supersteps: in iteration 1, each vertex v sends a message $\langle v\rangle$ to each neighbor $u \in \Gamma_{out}(v)$ to request for $a(u)$; then in iteration 2, each vertex u obtains the requesters v from the incoming messages, and sends the response message $\langle u, a(u)\rangle$ to each requester v; finally in iteration 3, each vertex v collects the incoming messages into $\widehat{\Gamma}_{out}(v)$.

2.2.2 PageRank

Next we present a Pregel algorithm for PageRank computation. Given a directed web graph $G = (V, E)$, where each vertex (page) v links to a list of pages $\Gamma_{out}(v)$, the problem is to compute the PageRank, $pr(v)$, of each vertex $v \in V$.

Pregel's PageRank algorithm [33] works as follows. In superstep 1, each vertex v initializes $pr(v) = 1/|V|$ and distributes the value $\langle pr(v)/d_{out}(v)\rangle$ to each out-neighbor of v. In superstep i ($i > 1$), each vertex v sums up the received values from its in-neighbors, denoted by sum, and computes $pr(v) = 0.15/|V| + 0.85 \times sum$. It then distributes $\langle pr(v)/d_{out}(v)\rangle$ to each of its out-neighbors.

2.2.3 Hash-Min

We next present a Pregel algorithm for computing connected components (CCs) in an undirected graph. We adopt the *Hash-Min* algorithm [37, 56]. Given a CC C, let us denote the set of vertices of C by $V(C)$, and define the ID of C to be $id(C) = \min\{id(v) : v \in V(C)\}$. We further define the *color* of a vertex v as $cc(v) = id(C)$, where $v \in V(C)$. *Hash-Min* computes $cc(v)$ for each vertex $v \in V$, and the idea is to broadcast the smallest vertex ID seen so far by each vertex v, denoted by $min(v)$. When the algorithm terminates, $min(v) = cc(v)$ for each vertex $v \in V$.

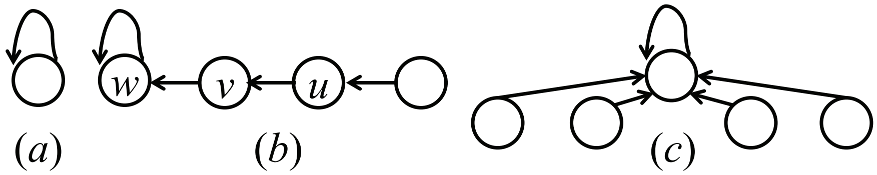


Figure 2.2: Forest structure of the S-V algorithm

We now describe the *Hash-Min* algorithm in Pregel framework. In superstep 1, each vertex v sets $min(v)$ to be $id(v)$, broadcasts $min(v)$ to all its neighbors, and votes to halt. In each subsequent superstep, each vertex v receives messages from its neighbors; let min^* be the smallest ID received, if $min^* < min(v)$, v sets $min(v) = min^*$ and broadcasts min^* to its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, all vertices have voted to halt and for each vertex v, we have $min(v) = cc(v)$.

2.2.4 The S-V Algorithm

The *Hash-Min* algorithm described in Section 2.2.3 requires $O(\delta)$ supersteps, which can be slow for computing CCs in large-diameter graphs. Another Pregel algorithm proposed in [56] for computing CCs takes $O(\log |V|)$ supersteps, by adapting Shiloach-Vishkin's (S-V) algorithm for the PRAM model [44]. This algorithm demonstrates how algorithm logic can generate a bottleneck vertex v even if $|\Gamma(v)|$ is small.

In the S-V algorithm, each vertex u maintains a pointer $D[u]$, which is initialized as u, forming a self loop as shown Figure 2.2(a). Throughout the algorithm, vertices are organized by a forest such that all vertices in a tree belong to the same CC. The tree definition is relaxed a bit here to allow the tree root w to have a self-loop (see Figures 2.2(b) and 2.2(c)), i.e., $D[w] = w$; while $D[v]$ of any other vertex v in the tree points to v's parent.

The S-V algorithm proceeds in rounds, and in each round, the pointers are updated in three steps (illustrated in Figure 2.3): (1)*tree hooking*: for each edge (u, v), if u's parent $w = D[u]$ is a tree root, hook w as a child of v's parent $D[v]$, i.e., set $D[D[u]] = D[v]$; (2)*star hooking*: for each edge (u, v), if u is in a star (see Figure 2.2(c) for an example of star), hook the star to v's tree as in Step (1), i.e., set $D[D[u]] = D[v]$; (3)*shortcutting*: for each vertex v, move vertex v and its descendants closer to the tree root, by hooking v to the parent of v's parent, i.e., setting $D[v] = D[D[v]]$. The algorithm ends when every vertex is in a star.

Due to the shortcutting operation, the S-V algorithm creates flattened trees (e.g., stars) with large fan-out towards the end of the execution. As a result, a vertex w may have many children u (i.e. $w = D[u]$), and each of these children u requests $w = D[u]$ for the value of $D[w]$. This renders w a bottleneck vertex. In particular, in the last round of the S-V

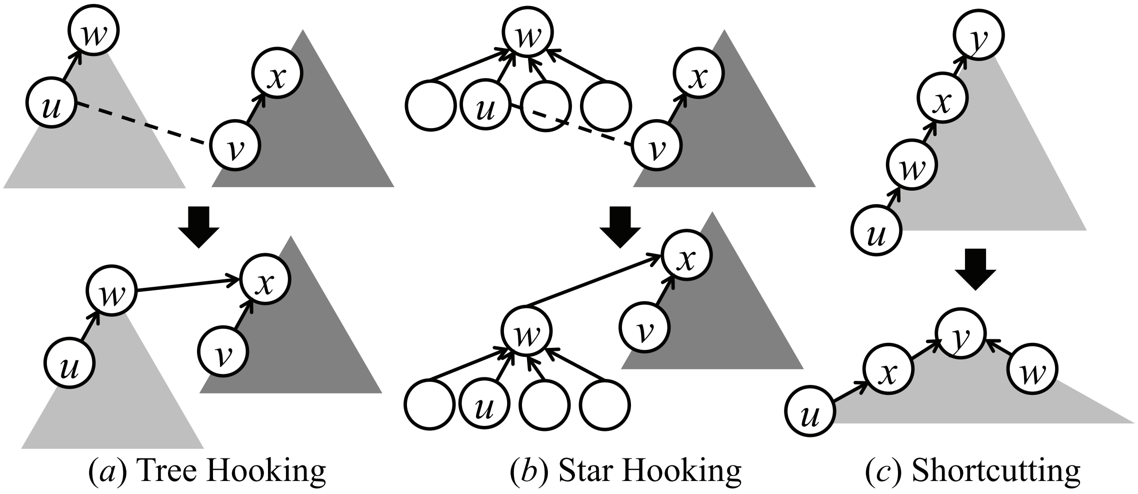


Figure 2.3: Key operations of the S-V algorithm

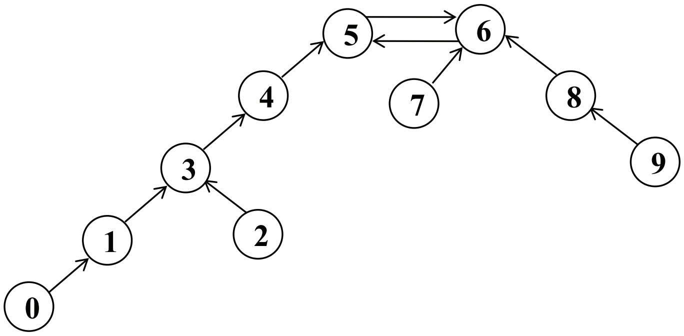


Figure 2.4: Conjoined Tree

algorithm, all vertices v in a CC C have $D[v] = id(C)$, and they all request for $D[w]$ from vertex $w = id(C)$. In the ordinary Pregel implementation, w receives $|V(C)|$ requests and sends $|C|$ responds, which leads to skewed workload when $|V(C)|$ is large.

2.2.5 Minimum Spanning Forest

The Pregel algorithm proposed by [41] for minimum spanning forest (MSF) computation serves as another example illustrating how algorithm logic can generate bottleneck vertices. This algorithm proceeds in iterations, where each iteration consists of three steps, which we describe below.

In Step (1), each vertex v picks an edge with the minimum weight. The vertices and their picked edges form disjoint subgraphs, each of which is a conjoined-tree: two trees with their roots joined by a cycle. Figure 2.4 illustrates the concept of a conjoined-tree, where the edges are those chosen in Step (1). The vertex with the smaller ID in the cycle of a conjoined-tree is called the supervertex of the tree (e.g., Vertex 5 is the supervertex in Figure 2.4), and the other vertices are called the subvertices.

In Step (2), each vertex finds the supervertex of the conjoined-tree it belongs to, which is accomplished by pointer jumping. Specifically, each vertex v maintains a pointer $D[v]$; suppose that v picks edge (v, u) in Step (1), then the value of $D[v]$ is initialized as u. Each vertex v then sends request to $w = D[v]$ for $D[w]$. Initially, the actual supervertex s (e.g. Vertex 5 in Figure 2.4) and its neighbor s' in the cycle (e.g. Vertex 6 in Figure 2.4) see that they have sent each other messages and detect that they are in the cycle. Vertex s then sets itself as the supervertex (i.e., sets $D[s] = s$) due to $s < s'$ before responding (while $D[s'] = s$ remains for Vertex s' since $s' > s$). For any other vertex v, it receives response $D[w]$ from $w = D[v]$ and updates $D[v]$ to be $D[w]$. This process is repeated until convergence, upon when $D[v]$ records the supervertex s for all vertices v.

In Step (3), each vertex v sends request to each neighbor $u \in \Gamma(v)$ for its supervertex $D[u]$, and removes edge (v, u) if $D[v] = D[u]$ (i.e., v and u are in the same conjoined-tree); v then sends the remaining edges (to vertices in other conjoined-trees) to the supervertex $D[v]$. After this step, all subvertices vertices are condensed into their supervertex, which constructs an adjacency list of edges to the other supervertices from those edges sent by its subvertices.

We consider an improved version of the above algorithm that applies the SEAS optimization of [41]. Specifically, instead of having the supervertex merge and store all cross-tree edges, the SEAS optimization stores the edges of a supervertex in a distributed fashion among all of its subvertices. As a result, if a supervertex s is merged into another supervertex, it has to notify its subvertices of the new supervertex they belong to. This is accomplished by having each vertex v send request to its supervertex $D[v] = s$ for $D[s]$. Since smaller conjoined-trees are merged into larger ones, a supervertex s may have many subvertices v towards the end of the execution, and they all request for $D[s]$ from s, rendering s a bottleneck vertex.

2.3 Overview of the Basic Pregel+ System

We now give an overview of the Pregel+ system, on which we implement our two message reduction techniques. In this section, we only describe the basic vertex-centric framework of Pregel+. We present the system extensions to support effective message reduction in Sections 2.4 and 2.5.

We use the term "worker" to represent a computing unit, which can be a machine or a thread/process in a machine. For ease of discussion, we assume that each machine runs only one worker but the concepts can be straightforwardly generalized.

2.3.1 Communication Framework of Pregel+

In Pregel+, each worker is simply an MPI (Message Passing Interface) process and communications among different processes are implemented using MPI's communication primitives. Each worker maintains a *message channel*, Ch_{msg}, for exchanging the vertex-to-vertex messages. In the *compute*() function, if a vertex sends a message msg to a target vertex v_{tgt}, the message is simply inserted into Ch_{msg}. Like in Google's Pregel, messages in Ch_{msg} are sent to the target workers in batch before the next superstep begins. Note that if a message msg is sent from worker M_i to vertex v_{tgt} in worker M_j, the ID of the target v_{tgt} should be sent along with msg, so that when M_j receives msg, it knows which vertex msg should be directed to.

The operation of the message channel Ch_{msg} is directly related to the communication cost and hence affects the overall performance of the system. We tested different ways of implementing Ch_{msg}, and the most efficient one is presented in Figure 2.5. We assume that a worker maintains N vertices, $\{v_1, v_2, \ldots, v_N\}$. The message channel Ch_{msg} associates each vertex v_i with an incoming message buffer I_i. When an incoming message msg_1 directed to vertex v_i arrives, Ch_{msg} looks up a hash table T_{in} for the incoming message buffer I_i using v_i's ID. It then appends msg_1 to the end of I_i. The lookup table T_{in} is static unless graph mutation occurs, in which case updates to T_{in} may be required. Once all incoming messages are processed, *compute*() is called for each active vertex v_i with the messages in I_i as the input.

A worker also maintains M *outgoing message buffers* (where M is the number of machines), one for each worker M_j in the cluster, denoted by O_j. In *compute*(), a vertex v_i may send a message msg_2 to another vertex with ID tgt. Let $hash(.)$ be the hash function that computes the worker ID of a vertex from its vertex ID, then the target vertex is in machine $M_{hash(tgt)}$. Thus, msg_2 (along with tgt) is appended to the end of the buffer $O_{hash(tgt)}$. Messages in each buffer O_j are sent to worker M_j in batch. If a combiner is used, the messages in a buffer O_j are first grouped (sorted) by target vertex IDs, and messages in each group are combined into one message using the combiner logic before sending.

We also tried other options of implementing Ch_{msg}, such as implementing O_j as a hash table with target vertex ID tgt as the key and the set of messages (or the currently combined message) directed to tgt as the value. We find that the implementation described in Figure 2.5 is consistently the most efficient one for various computing tasks over various

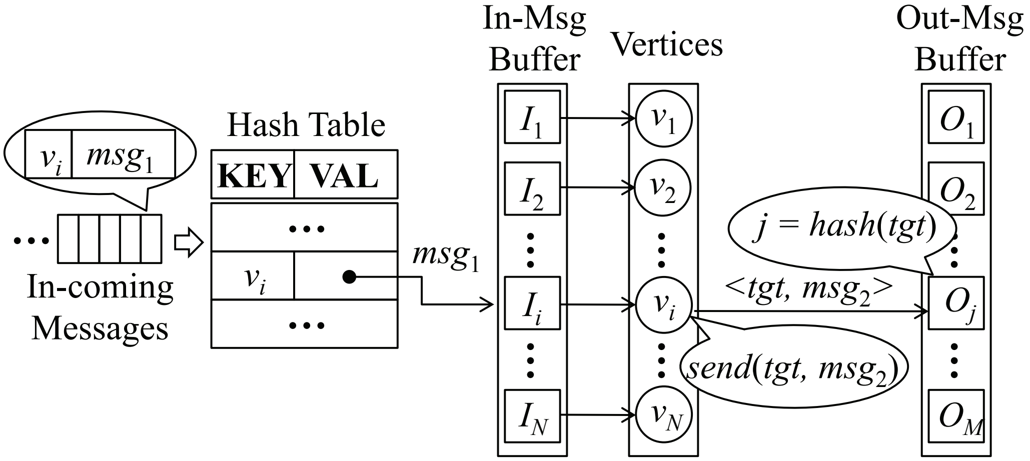


Figure 2.5: Illustration of Message Channel, Ch_{msg}

large real graphs.

2.3.2 Comparison with States-of-the-Arts

Why implementing the message reduction techniques on Pregel+? While the basic vertex-to-vertex messages are exchanged through Ch_{msg} (with combiner properly applied), messages exchanged by our two message reduction techniques are though two other message channels, Ch_{mir} and Ch_{req}, which will be described in the next two sections. Therefore, if Ch_{msg} is implemented to be slow, the performance improvement of Ch_{mir} and Ch_{req} would be exaggerated. To fairly evaluate the benefits of our message reduction techniques, they should be implemented on top of a Pregel-like system with a reasonably fast Ch_{msg}.

Many studies have compared the performance of existing graph-parallel systems recently [31, 19, 7, 42, 18, 12], including the state-of-the-arts like Pregel+ [56], Giraph [1], GraphLab [30] and GPS [40]. [31] shows that Pregel+ (where messages are passed only through Ch_{msg}) is consistently more efficient than Giraph and GraphLab, which implies that Pregel+ is a reasonable choice for implementing and evaluating our message reduction techniques. Neither Pregel+ nor GPS can beat each other in all algorithms on all datasets, but we choose Pregel+ instead of GPS because of the following reason: to avoid the complicated interaction between vertex mirroring and message combining, GPS simply does not perform sender-side message combining, and thus, it is not representative of a typical Pregel-like system.

We ran Hash-Min on *BTC* in GPS, the results of which are presented in Figure 2.6, where the taller blue bars (the short red bars) indicate the total number of messages sent by

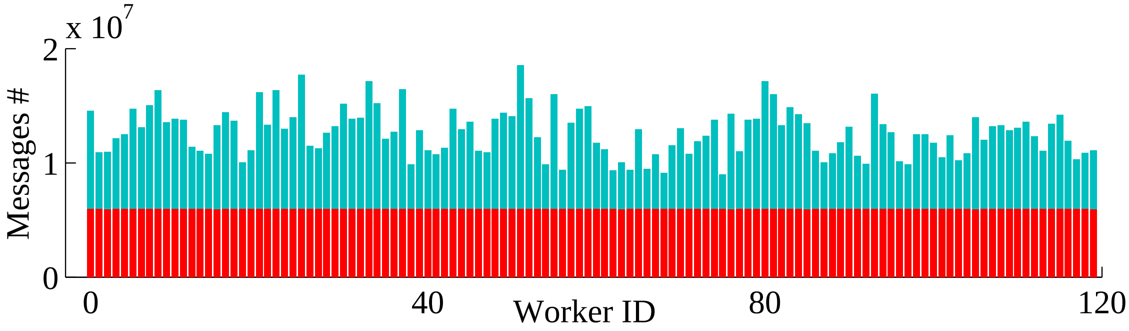


Figure 2.6: Message # of Each Worker: Hash-Min on BTC in GPS

each worker when GPS's LALP optimization is disabled (enabled by mirroring all vertices with degree at least 100)[1]. Compared with the results of Pregel+ shown in Figure 1.1, GPS sends many more messages per worker as message combining is not used; moreover, the effect of GPS's LALP is not as significant as Pregel+'s optimization that uses both vertex mirroring and message combining, which demonstrates the necessity of applying both techniques.

However, it is amazing that GPS is not much worse in most cases, and is sometimes even faster than Pregel+, even though more messages are exchanged. We studied the codes of GPS to explore the reason, which we explain below. GPS require that the vertex IDs should be integers that are contiguous starting from $0, \cdots, |V|$,[2] while other systems allow the vertex IDs to be of any user-specified type as long as a hash function is provided (for calculating the ID of the worker that each vertex resides in). As a result of the dense ID representation, each worker in GPS simply maintains the incoming message buffers of the vertices by an array, and when a worker receives a message targeted at vertex tgt, it is put into tgt's incoming message buffer (i.e., I_{tgt}) whose position in the array can be directly computed from tgt. On the other hand, systems like Pregel+ and Giraph need to look up I_{tgt} from a hash table using key tgt, and the cost is required for each message exchanged.

We remark that it is necessary to allow the vertex IDs to take arbitrary type, rather than to hard-code them as contiguous integers. For example, the Pregel algorithm of [56] for computing bi-connected components constructs an auxiliary graph from the input graph, and each vertex of the auxiliary graph corresponds to an edge (u, v) of the input graph. In this case, using integer pair as vertex ID saves the effort of relabeling the vertices of the

[1] We extend GPS to count the number of messages, and the updated codes are accessible from `https://github.com/pregel-like/gps`

[2] `https://sites.google.com/site/gpsdocumentation/home/input-graph-format`

auxiliary graph using integer IDs. Although we can easily implement the same message passing mechanism of GPS in Pregel+ to enjoy the better performance when vertex IDs are contiguous integers, it is out of the scope of this chapter which studies message reduction techniques for a general Pregel program.

2.4 The Mirroring Technique

The mirroring technique is designed to eliminate bottleneck vertices caused by high vertex degree. Given a high-degree vertex v, we construct a mirror for v in any worker in which some of v's neighbors reside. When v needs to send a message, e.g., the value of its attribute, $a(v)$, to its neighbors, v sends $a(v)$ to its mirrors and each mirror then forwards $a(v)$ to the neighbors of v that reside in the local worker of the mirror without any message passing.

Figure 2.7 illustrates the idea of mirroring. Assume that u_i is a high-degree vertex residing in worker machine M_1, and u_i has neighbors $\{v_1, v_2, \ldots, v_j\}$ residing in machine M_2 and neighbors $\{w_1, w_2, \ldots, w_k\}$ residing in machine M_3. Suppose that u_i needs to send a message $a(u_i)$ to the j neighbors in M_2 and k neighbors in M_3. Figure 2.7(a) shows how u_i sends $a(u_i)$ to its neighbors in M_2 and M_3 using Pregel's vertex-to-vertex message passing. In total, $(j + k)$ messages are sent, one for each neighbor. To apply mirroring, we construct a mirror for u_i in M_2 and M_3, as shown by the two squares (with label u_i) in Figure 2.7(b). In this way, u_i only needs to send $a(u_i)$ to the two mirrors in M_2 and M_3, and then, each mirror forwards $a(u_i)$ to u_i's neighbors locally in M_2 and M_3 without any network communication. This is illustrated in Figure 2.7(b). In total, only two messages are sent through the network, which not only tremendously reduces the communication cost, but also eliminates the imbalanced communication load caused by u_i.

We formalize the effectiveness of mirroring for message reduction by the following theorem.

Theorem 2.4.1. *Let $d(v)$ be the degree of a vertex v and M be the number of machines. Suppose that v is to deliver a message $a(v)$ to all its neighbors in one superstep. If mirroring is applied on v, then the total number of messages sent by v in order to deliver $a(v)$ to all its neighbors is bounded by* $\min\{M, d(v)\}$.

Proof. The proof follows directly from the fact that v only needs to send one message $a(v)$ to each of its mirrors in other machines and there can only be $\min\{M, d(v)\}$ mirrors of v. □

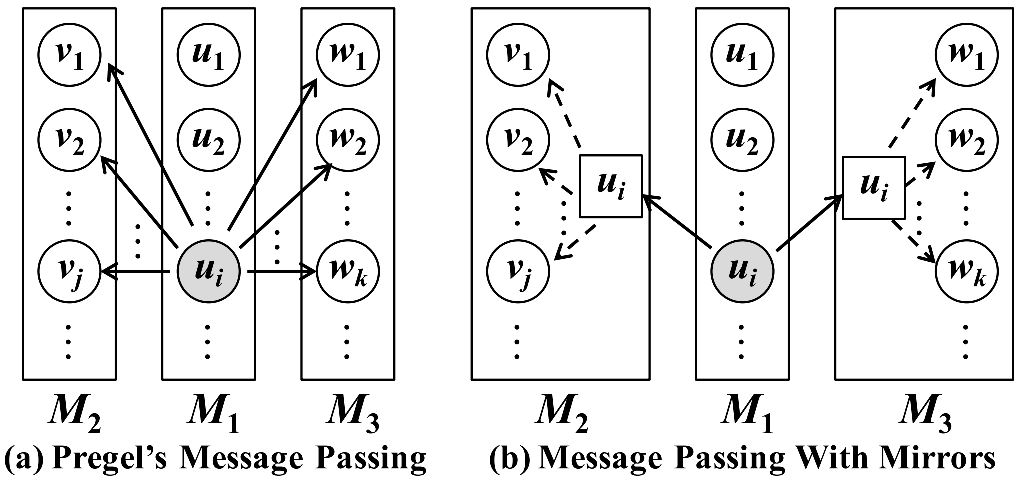

v_1
v_2
v_j
u_1
u_2
u_i
w_1
w_2
w_k
M_2
M_1
M_3
(a) Pregel's Message Passing
(b) Message Passing With Mirrors

Figure 2.7: Illustration of Mirroring

Mirroring Threshold. The mirroring technique is transparent to programmers. But we allow users to specify a mirroring threshold τ such that mirroring is applied to a vertex v only if $d(v) \geq \tau$ (we will see shortly that τ can be easily set following the result of Theorem 2.4.2). If a vertex has degree less than τ, it sends messages through the normal message channel Ch_{msg} as usual. Otherwise, the vertex only sends messages to its mirrors, and we call this message channel as the *mirroring message channel*, or Ch_{mir} in short. In a nutshell, a message is sent either through Ch_{msg} or Ch_{mir}, depending on the degree of the sending vertex.

Figure 2.8 illustrates the concepts of Ch_{msg} and Ch_{mir}, where we only consider the message passing between two machines M_1 and M_2. The adjacency lists of vertices u_1, u_2, u_3 and u_4 in M_1 are shown in Figure 2.8(a), and we consider how they send messages to their common neighbor v_2 residing in machine M_2. Assume that $\tau = 3$, then as Figure 2.8(b) shows, u_1, u_2 and u_3 send their messages, $a(u_1)$, $a(u_2)$ and $a(u_3)$, through Ch_{msg}, while u_4 sends its message $a(u_4)$ through Ch_{mir}.

Mirroring v.s. Message Combining. Now let us assume that the messages are to be applied with commutative and associative operations at the receivers' side, e.g., the message values are to be summed up as in PageRank computation. In this case, a combiner can be applied on the message channel Ch_{msg}. However, the *receiver-centric* message combining is not applicable to the *sender-centric* channel Ch_{mir}. For example, in Figure 2.8(b), when u_4 in M_1 sends $a(u_4)$ to its mirror in M_2, u_4 is not aware of the receivers (i.e., v_1, v_2, v_3 and v_4); thus, its message to v_2 cannot be combined with those messages from u_1, u_2 and u_3 that are also to be sent to v_2. In fact, u_4 only holds a list of the machines that

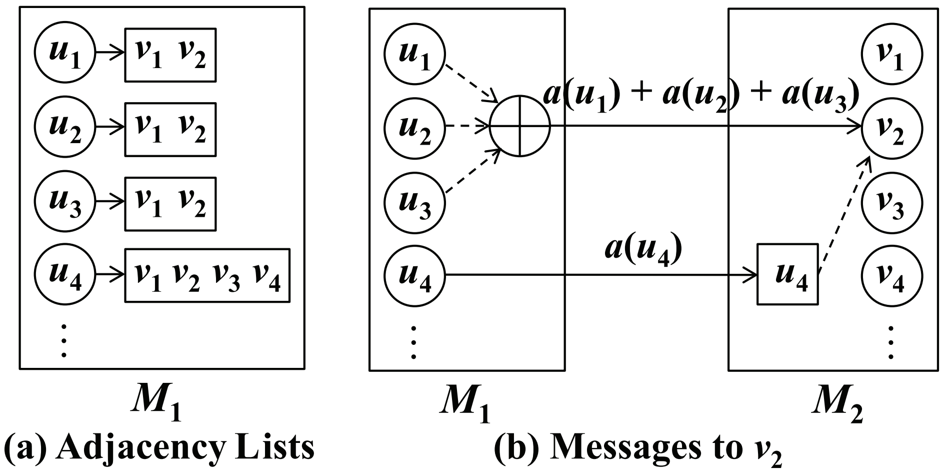


Figure 2.8: Mirroring v.s. Message Combining

contain u_4's neighbors, i.e. $\{M_2\}$ in this example, and u_4's neighbors v_1, v_2, v_3 and v_4 that are local to M_2 are connected by u_4's mirror in M_2.

It may appear that u_4's message to its mirror is wasted, because if we combine u_4's message with those messages from u_1, u_2 and u_3, then we do not need to send it through Ch_{mir}. However, we note that a high-degree vertex like u_4 often has many vertices in another worker machine, e.g., v_1, v_3 and v_4 in addition to v_2 in this example, and the message is not wasted since the message is also forwarded to v_3 and v_4, which are not the neighbors of any other vertex in M_1. In the following discussion, we further provide a theoretical analysis to show that mirroring is effective even when message combiner is used.

Choice of Mirroring Threshold. We now analyze the interplay between mirroring and message combining when combiner is applied, as detailed by the following theorem.

Theorem 2.4.2. *Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we assume that the vertex set is evenly partitioned among M machines (i.e., each machine holds n/M vertices). We further assume that the neighbors of a vertex in G are randomly chosen among V, and the average degree $deg_{avg} = m/n$ is a constant. Then, mirroring should be applied to a vertex v if v's degree is at least $(M \cdot \exp\{deg_{avg}/M\})$.*

Proof. Consider a machine M_i that contains a set of n/M vertices, $V_i = \{v_1, v_2, \ldots, v_{n/M}\}$, where each vertex v_j has ℓ_j neighbors for $1 \leq j \leq n/M$. Let us focus on a specific vertex v_j on M_i, and infer how large ℓ_j should be so that applying mirroring on v_j can reduce the overall communication even when a combiner is used.

Consider an application where all vertices send messages to all their neighbors in each superstep, such as in PageRank computation. Let $u \in \Gamma_{out}(v_j)$ be a neighbor of v_j. Then, if another vertex $v_k \in V_i \setminus \{v_j\}$ sends messages through Ch_{msg} and v_k also has u as its neighbor, then v_j's message to u is wasted since there already exists another combined message from M_i to u. We assume that in the worst case all vertices in $V_i \setminus \{v_j\}$ sends messages through Ch_{msg}. Since the neighbors of a vertex in G are randomly chosen among V, we have

$$\Pr\{u \in \Gamma_{out}(v_k)\} = \ell_k/n,$$

and therefore,

$$\begin{aligned} &\Pr\{v_j\text{'s message to } u \text{ is not wasted}\} \\ = &\prod_{v_k \in V_i \setminus \{v_j\}} \Pr\{u \notin \Gamma_{out}(v_k)\} \ = \prod_{v_k \in V_i \setminus \{v_j\}} \left(1 - \frac{\ell_k}{n}\right). \end{aligned}$$

We regard each ℓ_k as a random variable whose value is chosen independently from a degree distribution (e.g., power-law degree distribution) with expectation $E[\ell_k] = m/n = deg_{avg}$. Then, the expectation of the above equation is given by

$$\begin{aligned} &E\left[\prod_{v_k \in V_i \setminus \{v_j\}} \left(1 - \frac{\ell_k}{n}\right)\right] \ = \prod_{v_k \in V_i \setminus \{v_j\}} E\left[1 - \frac{\ell_k}{n}\right] \\ = &\prod_{v_k \in V_i \setminus \{v_j\}} \left(1 - \frac{E[\ell_k]}{n}\right) \ = \prod_{v_k \in V_i \setminus \{v_j\}} \left(1 - \frac{deg_{avg}}{n}\right) \\ \geq &\prod_{v_k \in V_i} \left(1 - \frac{deg_{avg}}{n}\right) \ = \left(1 - \frac{deg_{avg}}{n}\right)^{n/M}. \end{aligned}$$

For large graphs, we have

$$\begin{aligned} &\Pr\{v_j\text{'s message to } u \text{ is not wasted}\} \\ \approx &\lim_{n\to\infty} \left(1 - \frac{deg_{avg}}{n}\right)^{n/M} \ = \exp\left\{-\frac{deg_{avg}}{M}\right\}, \end{aligned}$$

where the last step is derived from the fact that $\lim_{n\to\infty}(1 - 1/n)^n = e^{-1}$.

According to the above discussion, the expected number of v_j's neighbors that are not the neighbors of any other vertex(es) in M_i is equal to $\ell_j \cdot \exp\{-deg_{avg}/M\}$. In other words, if mirroring is not used, v_j needs to send at least $\ell_j \cdot \exp\{-deg_{avg}/M\}$

messages that are not wasted. On the other hand, if mirroring is used, v_j sends at most M messages, one to each mirror. Therefore, mirroring reduces the number of messages if $\ell_j \cdot \exp\{-deg_{avg}/M\} \geq M$, or equivalently, $\ell_j \geq M \cdot \exp\{deg_{avg}/M\}$. To conclude, choosing $\tau = M \cdot \exp\{deg_{avg}/M\}$ as the degree threshold reduces the communication cost. □

Theorem 2.4.2 states that the choice of τ depends on the number of workers, M, and the average vertex degree, deg_{avg}. A cluster usually involves tens to hundreds of workers, while the average degree deg_{avg} of a large real world graph is mostly below 50. Consider the scenario where $M = 100$ and $deg_{avg} \leq 50$, then $\tau \leq 100e^{0.5}$=165. This shows that mirroring is effective even for vertices whose degree is not very high.

Mirror Construction. Pregel+ constructs mirrors for all vertices v with $\Gamma_{out}(v) \geq \tau$ after the input graph is loaded and before the iterative computation, although mirror construction can also be pre-computed offline like GraphLab's ghost construction. Specifically, the neighbors in v's adjacency list Γ_{out} is grouped by the workers in which they reside. Each group is defined as $N_i = \{u \in \Gamma_{out}(v) \mid hash(u) = M_i\}$. Then, for each group N_i, v sends $\langle v; N_i \rangle$ to worker M_i, and M_i constructs a mirror of v with the adjacency list N_i locally in M_i. Each vertex $v_j \in N_i$ also stores the address of v_j's incoming message buffer I_j so that messages can be directly forwarded to v_j.

During graph computation, a vertex v sends message $\langle v, a(v) \rangle$ to its mirror in worker M_i. On receiving the message, M_i looks up v's mirror from a hash table using v's ID (similar to T_{in} described in Section 2.3). The message value $a(v)$ is then forwarded to the incoming message buffers of v's neighbors locally in M_i.

Handling Edge Fields. There are some minor differences from Pregel's original interface. In Pregel's original interface, a vertex calls *send_msg*(tgt, msg) to send an arbitrary message msg to a target vertex tgt. When mirroring is applied, a vertex v sends a message containing the value of its attribute $a(v)$ to all its neighbors by calling *broadcast*$(a(v))$ instead of calling *send_msg*$(u, a(v))$ for each neighbor $u \in \Gamma_{out}(v)$.

Consider the algorithms described in Section 2.2. For PageRank, a vertex v simply calls *broadcast*$(pr(v)/|\Gamma_{out}(v)|)$; while for *Hash-Min*, v calls *broadcast*$(min(v))$. However, there are applications where the message value is not only decided by the sender vertex v's state, but also by the edge that the message is sent along. For example, in Pregel's algorithm for single-source shortest path (SSSP) computation [33], a vertex sends $(d(v) + \ell(v, u))$ to each neighbor $u \in \Gamma_{out}(v)$, where $d(v)$ is an attribute of v estimating the distance from the source, and $\ell(v, u)$ is an attribute of its out-edge (v, u) indicating the edge length.

To support applications like SSSP, Pregel+ requires that each edge object supports a function *relay*(msg), which specifies how to update the value of msg before msg is added to the incoming message buffer I_i of the target vertex v_i. If msg is sent through Ch_{msg}, *relay*(msg) is called on the sender-side before sending. If msg is sent through Ch_{mir}, *relay*(msg) is called on the receiver-side when the mirror forwards msg to each local neighbor (since the edge field is maintained by the mirror). For example, in Figure 2.8, *relay*(msg) is called when msg is passed along a dashed arrow.

By default, *relay*(msg) does not change the value of msg. To support SSSP, a vertex v calls *broadcast*$(d(v))$ in *compute*(), and meanwhile, the function *relay*(msg) is overloaded to add the edge length $\ell(v, u)$ to msg, which updates the value of msg to the desired value $(d(v) + \ell(v, u))$.

Summary of Contributions. Both GraphLab and GPS do not use message combining, and therefore, their vertex mirroring techniques are not representative of a general Pregel program. As far as we know, this is the first work that considers both vertex mirroring and message combining. We identified the conflicts of vertex mirroring and message combining in reducing message number, and gave a cost model for automatically selecting vertices for mirroring so as to minimize the number of messages. As we shall see in our experiments in Section 2.6.2, the theoretical mirroring threshold is quite accurate. Moreover, we also cope with the case where the message value depends on the edge field, which is not supported by GPS's LALP technique.

2.5 The Request-Respond Paradigm

In Sections 2.2.4 and 2.2.5, we have demonstrated that bottleneck vertices can be generated by algorithm logic even if the input graph has no high-degree vertices. For handling such bottleneck vertices, the mirroring technique proposed in Section 2.4 is not effective. To this end, we design our second message reduction technique, i.e. to extend the basic Pregel framework with a new *request-respond* functionality.

We illustrate the concept using the applications described in Section 2.2. Using the request-respond API, attribute broadcast in Section 2.2.1 is straightforward to implement: in superstep 1, each vertex v sends requests to each neighbor $u \in \Gamma_{out}(v)$ for $a(u)$; in superstep 2, the vertex v simply obtains $a(u)$ responded by each neighbor u, and constructs $\widehat{\Gamma}_{out}(v)$. Similarly, in the S-V algorithm of Section 2.2.4, when a vertex v needs to obtain $D[w]$ from vertex $w = D[v]$, it simply sends a request to w so that the value $D[w]$ can be used in the next superstep; in the MSF algorithm of Section 2.2.5, a vertex v simply sends a request to its supervertex $D[v] = s$ so that the value $D[s]$ can be used to update $D[v]$ in the next superstep.

Request-Respond Message Channel. We now explain in detail how Pregel+ supports

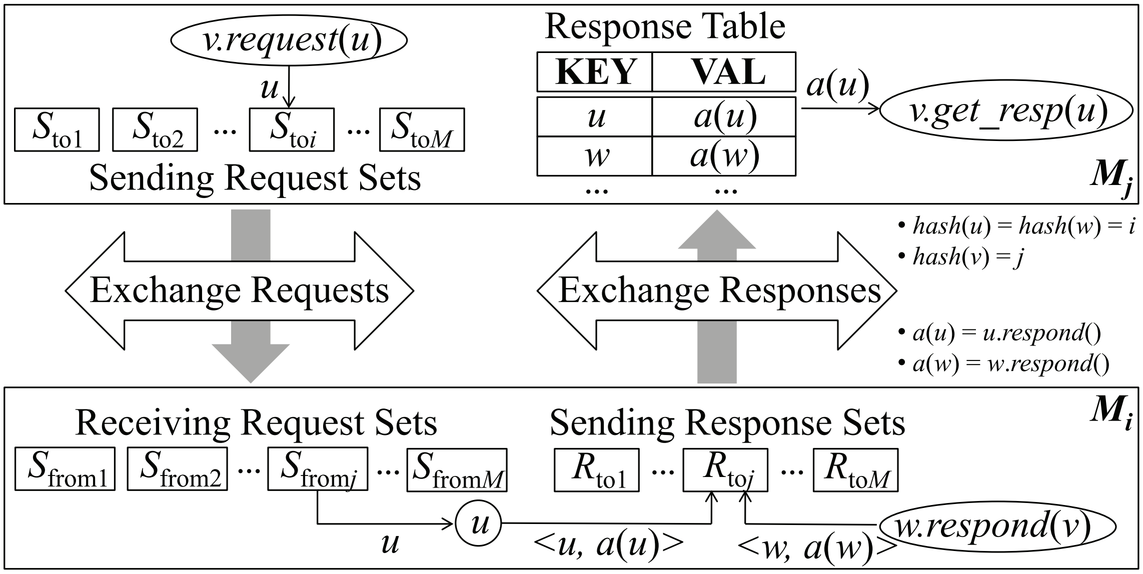


Figure 2.9: Illustration of request-respond paradigm

the request-respond API. Unlike the mirroring technique which slightly changes the programming interface of Pregel, the request-respond paradigm supports all the functionality of Pregel. In addition, it compensates the vertex-to-vertex message channel Ch_{msg} with a *request-respond message channel*, denoted by Ch_{req}.

Figure 2.9 illustrates how requests and responses are exchanged between two machines M_i and M_j through Ch_{req}. Specifically, each machine maintains M request sets, where M is the number of machines, and each request set $S_{\text{to}\,k}$ stores the requests to vertices in machine M_k. In a superstep, a vertex v in machine M_j may call *request*(u) in its *compute*() function to send request to vertex u for its attribute value $a(u)$ (which will be used in the next superstep). Let $hash(u) = i$, then the requested vertex u is in machine M_i, and hence u is added to the request set $S_{\text{to}\,i}$ of M_j. Although many vertices in M_j may send request to u, only one request to u will be sent from M_j to M_i since $S_{\text{to}\,i}$ is a (hash) set that prevents redundant elements.

After *compute*() is called for all active vertices, the vertex-to-vertex messages are first exchanged through Ch_{msg}. Then, each machine sends each request set $S_{\text{to}\,k}$ to machine M_k. After the requests are exchanged, each machine receives M request sets, where set $S_{\text{from}\,k}$ stores the requests sent from machine M_k. In the example shown in Figure 2.9, u is contained in the set $S_{\text{from}j}$ in machine M_i, since vertex v in machine M_j sent request to u.

Then, a response set $R_{\text{to}k}$ is constructed for each request set $S_{\text{from}\,k}$ received, which is to be sent back to machine M_k. In our example, given the requested vertex $u \in S_{\text{from}j}$, u

calls a user-specified function *respond*() to return its specified attribute $a(u)$, and adds the entry $\langle u, a(u) \rangle$ to the response set $R_{\text{to}j}$.

Once the response sets are exchanged, each machine constructs a hash table from the received entries. In the example shown in Figure 2.9, the entry $\langle u, a(u) \rangle$ is received by machine M_j since it is in the response set $R_{\text{to}j}$ in machine M_i. The hash table is available for the next superstep, where vertices can access their requested value in their *compute*() function. In our example, vertex v in machine M_j may call *get_resp*(u) in the next superstep, which looks up u's attribute $a(u)$ from the hash table.

The following theorem shows the effectiveness of the request-respond paradigm for message reduction.

Theorem 2.5.1. *Let $\{v_1, v_2, \ldots, v_\ell\}$ be the set of requesters that request the attribute $a(u)$ from a vertex u. Then, the request-respond paradigm reduces the total number of messages from 2ℓ in Pregel's vertex-to-vertex message passing framework to $2\min(M, \ell)$, where M is the number of machines.*

Proof. The proof follows directly from the fact that each machine sends at most 1 request to u even though there may be more than 1 requester in that machine, and that at most 1 respond from u is sent to each machine that makes a request to u, and that there are at most $\min(M, \ell)$ machines that contain a requester. □

In the worst case, the request-respond paradigm uses the same number of messages as Pregel's vertex-to-vertex message passing. In practice, however, many Pregel algorithms (e.g., those described in Sections 2.2.4 and 2.2.5) have bottleneck vertices with a large number of requesters, leading to imbalanced workload and prolonged elapsed running time. In such cases, our request-respond paradigm effectively bounds the number of messages to the number of machines containing the requesters and eliminates the imbalanced workload.

Explicit Responding. In the above discussion, a vertex v simply calls *request*(u) in one superstep, and it can then call *get_resp*(u) in the next superstep to get $a(u)$. All the operations including request exchange, response set construction, response exchange, and response table construction are performed by Pregel+ automatically and are thus transparent to users. We call the above process as implicit responding, where a responder does not know the requester until a request is received.

When a responder w knows its requesters v, w can explicitly call *respond*(v) in *compute*(), which adds $\langle w, w.\textit{respond}() \rangle$ to the response set $R_{\text{to}\,j}$ where $j = hash(v)$. This process is also illustrated in Figure 2.9. Explicit responding is more cost-efficient since there is no need for request exchange and response set construction.

Explicit responding is useful in many applications. For example, to compute PageRank on an undirected graph, a vertex v can simply call *respond*(u) for each $u \in \Gamma_{out}(v)$ to push $a(v) = pr(v)/|\Gamma_{out}(v)|$ to v's neighbors; this is because in the next superstep, vertex u knows its in-neighbors $\Gamma_{in}(u) = \Gamma_{out}(u)$, and can thus collect their responses. Similarly, in attribute broadcast, if the input graph is undirected, each vertex v can simply push its attribute $a(v)$ to its neighbors. Note that data pushing by explicit responding requires less messages than by Pregel's vertex-to-vertex message passing, since responds are sent to machines (more precisely, their response tables) rather than individual vertices.

Programming Interface. The vertex class that supports the request-respond API requires users to specify an additional template argument <*R*>, which indicates the type of the attribute value that a vertex responds.

In *compute*(), a vertex can either pull data from another vertex v by calling *request*(v), or push data to v by calling *respond*(v). The attribute value that a vertex returns is defined by a user-specified abstract function *respond*(), which returns a value of type <*R*>. Like *compute*(), one may program *respond*() to return different attributes of a vertex in different supersteps according to the algorithm logic of the specific application. Finally, a vertex may call *get_resp*(v) in *compute*() to get the attribute of v, if it is pushed into the response table in the previous superstep.

2.6 Experimental Results

In this section, we evaluate the performance of Pregel+ and compare it with Giraph 1.0.0 [1] and GraphLab 2.2 (which includes all the features of PowerGraph [17]). We ran our experiments on a cluster of 16 machines, each with 24 processors (two Intel Xeon E5-2620 CPU) and 48GB RAM. One machine is used as the master that runs only one worker, while the other 15 machines act as slaves running multiple workers. The connectivity between any pair of nodes in the cluster is 1Gbps.

We used five large real-world datasets, which are shown in Figure 2.10: (1)*WebUK*[1]: a web graph generated by combining twelve monthly snapshots of the .uk domain collected for the DELIS project; (2)*LiveJournal*[2]: a bipartite network of LiveJournal users and their group memberships; (3)*Twitter*[3]: Twitter who-follows-who network based on a snapshot taken in 2009; (4)*BTC*[4]: a semantic graph converted from the Billion Triple Challenge

[1] http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05

[2] http://konect.uni-koblenz.de/networks/livejournal-groupmemberships

[3] http://konect.uni-koblenz.de/networks/twitter_mpi

[4] http://km.aifb.kit.edu/projects/btc-2009/

Data	Type	|V|	|E|	Avg Deg	Max Deg	# of Workers Per Slave
WebUK	directed	133.6M	5507.7M	41.21	22,429	10
LiveJournal	directed	10.7M	224.6M	21.01	1,053,676	2
Twitter	directed	52.6M	1963.3M	37.34	779,958	8
BTC	undirected	164.7M	772.8M	4.69	1,637,619	8
USA Road	undirected	23.9M	58.3M	2.44	9	4

Figure 2.10: Datasets (M=1,000,000)

2009 RDF dataset; (5)*USA*[5]: the USA road network.

LiveJournal, *Twitter* and *BTC* have skewed degree distribution; *WebUK*, *LiveJournal* and *Twitter* have relatively high average degree; *USA* and *WebUK* have a large diameter.

In all the experiments, we run 1 worker on the master machine, and for each of the 15 slave machines, we run multiple workers for different datasets depending on the size of the datasets (less workers are used for smaller datasets). We give the number of workers running on each slave machine for each dataset in the last column of the table shown in Figure 2.10.

2.6.1 Pregel+ Implementation

We make Pregel+ open-source. All the system source codes, as well as the source codes of the applications discussed in this chapter, can be found in `http://www.cse.cuhk.edu.hk/pregelplus`.

Pregel+ is 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+ communicates with HDFS through libhdfs, a JNI based C API for HDFS. Each worker is simply an MPI process and communications are implemented using MPI communication primitives. While one may deploy Pregel+ with any Hadoop and MPI version, we use Hadoop 1.2.1 and MPICH 3.0.4 in our experiments. All programs are compiled using GCC 4.4.7 with -O2 option enabled.

2.6.2 Effect of Mirroring

Figure 2.11 compares the performance of Pregel+ in basic mode with that when mirroring is used, where we test the mirroring thresholds 1, 10, 100, 1000, and the one computed by

[5] http://www.dis.uniroma1.it/challenge9/download.shtml

		Pregel+	Pregel+ with Veretx Mirroring					Giraph	GraphLab		GPS	GPS + LALP
			Mirroring Thresholds									
			1	10	100	1000	Theoretical		Sync	Async		
PageRank over *WebUK*	Comp. Time	*2669 s**	5603 s	5561 s	3475 s	2784 s	**199** : 2935 s	4834 s	4262 s	-	3909 s	4020 s
	Mir. Build	-	162.29 s	143.00 s	46.68 s	32.79 s	26.66 s	-	-	-	-	663.34 s
	# of Msgs	120107 M	319614 M	314212 M	168317 M	*119889 M**	134734 M	-	-	-	487285 M	377687 M
PageRank over *Twitter*	Comp. Time	1575 s	1621 s	1648 s	1177 s	1381 s	**165** : ***1048 s***	1567 s	1762 s	-	1343 s	*750.11 s**
	Mir. Build	-	40.74 s	41.34 s	24.13 s	8.63 s	14.31 s	-	-	-	-	74.95 s
	# of Msgs	62276 M	68430 M	65770 M	40980 M	48616 M	*38873 M**	-	-	-	17471 M	78904 M
PageRank over *Livejournal*	Comp. Time	316.26 s	251.98 s	255.26 s	***212.35 s***	243.32 s	**62** : 216.05 s	312 s	662 s	-	316.45 s	*197.28 s**
	Mir. Build	-	9.95 s	7.72 s	3.75 s	1.07 s	3.94 s	-	-	-	-	8.17 s
	# of Msgs	6429 M	5949 M	*3949 M**	4209 M	5162 M	4359 M	-	-	-	21563 M	9665 M
Hash-Min over *BTC*	Comp. Time	26.97 s	29.95 s	15.53 s	*9.55 s**	10.69 s	**126** : 9.85 s	93 s	83 s	155 s	37.99 s	33.00 s
	Mir. Build	-	20.74 s	6.63 s	5.92 s	5.56 s	5.41 s	-	-	-	-	3.52 s
	# of Msgs	1189 M	1294 M	259.4 M	126.1 M	152.4 M	*122.5 M**	-	-	-	1525 M	716.4 M
Hash-Min over *USA*	Comp. Time	546.86 s	542.69 s	-	-	-	-	5714 s	2982 s	627 s	1205 s	-
	Mir. Build	-	4.52 s	-	-	-	-	-	-	-	-	-
	# of Msgs	8353 M	8305 M	-	-	-	-	-	-	-	8485 M	-

Figure 2.11: Effects of mirroring ($\star$: best result, M = million)

Theorem 2.4.2. We also include the performance of other systems (i.e., Giraph, GraphLab and GPS) for reference, and for GPS, we test both its basic mode and the version with LALP enabled. Following [19], we fix the threshold of LALP as 100, which is found to work well overall. Combiners are used for Pregel+ and Giraph. For Pregel+ with mirroring and GPS with LALP, we also report their mirror construction time. The algorithms involved are PageRank which we run on the three directed graph in Figure 2.10, and Hash-Min which we run on the two undirected graph in Figure 2.10. Note that mirroring is not applicable to the other algorithms described in Section 2.2. We do not run GraphLab in asynchronous mode for PageRank since its convergence condition is different from the synchronous version. Also note that Giraph and GraphLab do not report the number of messages sent during the computation, and thus, we cannot show the figures. Finally, since the maximum vertex degree is only 9 on *USA*, we do not show the results of mirroring with large thresholds.

In all the experiments, Pregel+ and GPS are consistently faster than Giraph and GraphLab even when running in basic mode, which verifies that our Ch_{msg} is already very fast, and thus, Pregel+ is a fair choice to evaluate the performance improvement of Ch_{mir}. GPS is sometimes faster than Pregel+ due to its narrower design, as we have discussed 2.3.2. However, the smallest number of messages sent during the computation is achieved by Pregel+ in all the experiment due to its usage of both message combiner and vertex mirroring (note that GPS does not use message combiner).

In the first experiment which computes PageRank on *WebUK*, the fastest performance is achieved by Pregel+'s basic mode. Only with mirroring threshold $\tau = 1000$ does vertex mirroring slightly decrease the number of messages, and yet the performance is still slower than that of the basic mode. This is because messages sent through Ch_{mir} are intercepted

		Pregel+		Pregel+ with Mir.		Giraph	
		Combiner	No Combiner	Combiner	No Combiner	Comb.	No Comb.
PageRank over *WebUK*	Time	*2669 s**	7732 s	2784 s	6482 s	4834 s	5817 s
	Msg #	120107 M	490184 M	*119889 M**	412198 M	-	-
PageRank over *Twitter*	Time	1575 s	3131 s	*1048 s**	1710 s	1567 s	1800 s
	Msg #	62276 M	174730 M	*38873 M**	78055 M	-	-
PageRank over *LiveJournal*	Time	316.26 s	541.53 s	*212.35 s**	314.76 s	311.50 s	364.44 s
	Msg #	6429 M	21563 M	*4359 M**	9644 M	-	-
Hash-Min over *BTC*	Time	26.97 s	44.28 s	*9.55 s**	27.91 s	93.13 s	107.56 s
	Msg #	1189M	2419 M	*126.1 M**	1309 M	-	-
Hash-Min over *USA*	Time	546.86 s	546.66 s	542.69 s	*531.90 s**	5714 s	5997 s
	Msg #	8353 M	8485 M	*8305 M**	8305 M	-	-

Figure 2.12: Effects of message combiner

by mirrors which incurs additional computation cost, and hence the throughput of Ch_{mir} is less than that of Ch_{msg}. Since the vertex degree of *WebUK* is relatively small and not very biased, mirroring does not significantly reduce the number of messages, and thus, the additional cost of Ch_{mir} is not paid off.

For the next three experiments, the graphs involved all have power-law degree distribution, and mirroring significantly reduces the number of messages. The theoretical threshold achieves both the shortest computation time and the smallest message number on *Twitter* and *BTC*, while its performance (i.e. 216.05 s) is very close to that of the best threshold tested (i.e., 212.35 s). In fact, even on *WebUK*, the theoretical threshold achieves performance not much worse than the best performance. This verifies that mirroring with the threshold automatically computed from Theorem 2.4.2 consistently gives reasonable performance.

Overall, mirroring significantly improves the performance of Pregel+ on power-law graphs, while LALP just slight improves the performance of GPS. For example, when computing Hash-Min on *BTC*, our theoretical threshold improves the performance of Pregel+ from 26.97 s to 9.85 s, while LALP only improves the performance of GPS from 37.99 s to 33 s. This shows the importance of using both mirroring and message combining.

	Pregel+	ReqResp	Giraph	GPS	Pregel+	ReqResp	Giraph	GPS
	Attribute Broadcast on *WebUK*				**S-V on *USA***			
Time	178.4 s	84.53 s	169.28 s	83.71 s*	261.93 s	137.69 s*	690 s	189.77 s
Msg #	11015 M	2699 M*	–	10950 M	6598 M	3789 M*	–	6598M
	Attribute Broadcast on *BTC*				**S-V on *BTC***			
Time	16.33 s	13.31 s	54.76 s	8.69 s*	408.78 s	190.55 s*	1531 s	286.22 s
Msg #	772.8 M	393.2 M*	–	772.8 M	22393 M	11232 M*	–	22393M
	Attribute Broadcast on *LiveJournal*				**Minimum Spanning Forest on *USA***			
Time	11.66 s	9.09 s	11.56 s	6.43 s*	19.95 s*	25.20 s	259.63 s	85.15 s
Msg #	449.2 M	131.9 M*	–	449.2 M	387.1 M	162.2 M*	–	387.1 M
	Attribute Broadcast on *Twitter*				**Minimum Spanning Forest on *BTC***			
Time	59.84 s	29.65 s*	71.35 s	29.93 s	83.36 s	36.56 s*	350.15 s	209.92 s
Msg #	3927 M	1396 M*	–	3927 M	2424 M	1110 M*	–	2424 M

Figure 2.13: Effects of the Request-Respond Paradigm

2.6.3 Performance of Message Combining

While Pregel [33] promotes message combiner as an effective tool of message reduction, [40] claims that message combining is not effective in GPS. To evaluate the effectiveness of message combining, we run PageRank and Hash-Min in Pregel+ and Giraph, both with and without message combiner. We do not use GraphLab and GPS since they do not support message combiner. We report the results in Figure 2.12, which clearly shows that applying combiner significantly improves the performance. For example, when computing *PageRank* on *WebUK*, Pregel+ takes only 2669.36 s and exchanges only 120107 million messages when combiner is used, while it takes 7732 s and exchanges 490184 million messages if there is no combiner. The only outlier is *USA*, where the fastest performance is achieved when mirroring is used and message combiner is not used. This is because vertices in *USA* have very low degree, rendering message combining ineffective; the computational overhead of message combining is not paid off by the reduced message number.

2.6.4 Effect of the Request-Respond Paradigm

Figure 2.13(b) compares the performance of Pregel+ in basic mode with the version that uses the request-respond paradigm. We test the three algorithms of Section 2.2 to which the request-respond paradigm is applicable: attribute broadcast, *S-V* and minimum spanning forest. We also include Giraph and GPS for reference, while we do not include

GraphLab since the algorithms do not easily translate to GraphLab. For example, in *S-V* and minimum spanning forest, a vertex v needs to communicate with a non-neighbor $D[v]$, which cannot be done in GraphLab.

In all the experiment, *ReqResq* the request-respond paradigm effectively reduces the number of messages. For example, in attribute broadcast on *WebUK*, *ReqResq* reduces the message number from 11015 million to only 2699 million. GPS is sometimes faster due to its narrower design; otherwise, *ReqResq* almost always achieves the best performance. The only exception is when computing minimum spanning forest on *USA*, where Pregel+'s basic mode is the fastest (i.e., 19.95 s). This is because vertices in *USA* have very low degree, rendering the request-respond paradigm ineffective. The additional computational overhead is not paid off by the reduced message number.

2.7 Conclusions

We identified two scenarios that lead to performance bottlenecks when using existing distributed graph computing systems. The first scenario is due to skewed degree distribution in many massive real-world graphs and the second scenario is as a result of program logic, both of which create imbalanced communication workload among different machines in a cluster. To reduce the amount of communication and eliminate communication bottlenecks, two message reduction techniques were presented in this chapter: (1)mirroring, which eliminates bottlenecks caused by high vertex degree, and is transparent to programming, and (2)a new request-respond paradigm, which eliminates bottlenecks caused by program logic, and simplifies the programming of many Pregel algorithms. Our experiments on large real-world graphs verify the effectiveness of our two techniques in reducing the communication cost and overall computation time.

Chapter 3

Blogel Architecture

The rapid growth in the volume of many real-world graphs (e.g., social networks, web graphs, and spatial networks) has led to the development of various vertex-centric distributed graph computing systems in recent years. However, real-world graphs from different domains have very different characteristics, which often create bottlenecks in vertex-centric parallel graph computation. We identify three such important characteristics from a wide spectrum of real-world graphs, namely (1)skewed degree distribution, (2)large diameter, and (3)(relatively) high density. Among them, only (1) has been studied by existing systems, but many real-world power-law graphs also exhibit the characteristics of (2) and (3). In this chapter, we propose a block-centric framework, called Blogel, which naturally handles all the three adverse graph characteristics. Blogel programmers may think like a block and develop efficient algorithms for various graph problems. We propose parallel algorithms to partition an arbitrary graph into blocks efficiently, and block-centric programs are then run over these blocks. Our experiments on large real-world graphs verified that Blogel is able to achieve orders of magnitude performance improvements over the state-of-the-art distributed graph computing systems.

The rest of the chapter is organized as follows. Section 3.1 reviews related systems. Section 3.2 illustrates the merits of block-centric computing. Section 3.3 gives an overview of the Blogel framework, Section 3.4 presents Blogel's programming interface, and Section 3.5 discusses algorithm design in Blogel. We describe our partitioners in Section 3.6 and present performance results in Section 3.7. Finally, we conclude this chapter in Section 3.8.

3.1 Background and Related Work

We first define some basic graph notations and discuss the storage of a graph in distributed systems.

Notations. Given an undirect graph $G = (V, E)$, we denote the neighbors of a vertex $v \in V$ by $\Gamma(v)$; if G is directed, we denote the in-neighbors (out-neighbors) of v by $\Gamma_{in}(v)$ ($\Gamma_{out}(v)$). Each $v{\in}V$ has a unique integer ID, denoted by $id(v)$. The diameter of G is denoted by $\delta(G)$, or simply δ when G is clear from the context.

Graph storage. We consider a shared-nothing architecture where data is stored in a *distributed file system (DFS)*, such as *Hadoop's DFS (HDFS)*. We assume that a graph is stored as a distributed file in HDFS, where each line records a vertex and its adjacency list. A distributed graph computing system involves a cluster of k workers, where each worker w_i holds and processes a batch of vertices in its main memory. Here, "worker" is a general term for a computing unit, and a machine can have multiple workers in the form of threads/processes. A job is processed by a graph computing system in three phases: **(1)loading**: each worker loads a portion of vertices from HDFS into main-memory; the workers then exchange vertices through the network (by hashing over vertex ID) so that each worker w_i finally holds all and only those vertices assigned to w_i; **(2)iterative computing**: in each iteration, each worker processes its own portion of vertices sequentially, while different workers run in parallel and exchange messages. **(3)dumping**: each worker writes the output of all its processed vertices to HDFS. Most existing graph-parallel systems follow this procedure.

3.1.1 Pregel's Computing Model

Pregel logic. Pregel [33] is designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to workers, where each vertex is associated with its adjacency list. A program in Pregel implements a user-defined *compute*() function and proceeds in iterations (called *supersteps*). In each superstep, the program calls *compute*() for each active vertex. The *compute*() function performs the user-specified task for a vertex v, such as processing v's incoming messages (sent in the previous superstep), sending messages to other vertices (for the next superstep), and making v vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Pregel also supports message combiner. For example, if there are k numerical messages in worker w_i to be sent to a vertex v in worker w_j and suppose that only the sum of the message values matters, then user can implement a *combine*() function to sum up the message values and deliver only the sum to v in w_j, thus reducing the number of messages to be buffered and transmitted. Pregel also supports aggregator, which is useful 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.

Open-source Pregel-like systems. Since Google's Pregel is proprietary, many open-source systems have been developed based on Pregel's computing model. Most open-source Pregel counterparts are implemented in Java, such as Giraph [1] and GPS [40]. However, since object deletion is handled by Java's Garbage Collector (GC), if workers maintain a huge amount of vertex/edge objects in main memory, GC needs to track a lot of objects and its overhead can severely degrade the system performance. To decrease the number of objects being maintained, Java-based Pregel systems maintain vertices in their binary representation in main memory. For example, Giraph organizes vertices as main memory pages, where each page is simply a byte array object that holds the binary representation of many vertices. As a result, a vertex needs to be deserialized from the page holding it before calling *compute*(), and after *compute*() completes, the updated vertex needs to be serialized back to its page. Usually, the serialization cost is more expensive than *compute*() itself, especially if the adjacency list is long. The serialization problem is exacerbated in a block-centric graph-parallel system due to the larger size of blocks. Therefore, we implement Blogel in C++ to completely eliminate this problem.

Graph partitioning. Vertex placement rules that are more sophisticated than hashing were studied in [45], which aims at minimizing the number of cross-worker edges while ensuring that workers hold approximately the same number of vertices. However, their method requires expensive preprocessing but the performance gain is limited (e.g., only a speed up of 18%–39% for running PageRank).

A recent system, Giraph++ [49], extends Giraph to support a graph-centric programming paradigm that opens up the block structure to users. However, Giraph++'s programming paradigm is still vertex-centric since it does not allow a block to have its own states like a vertex. Instead, each block is treated as two sets of vertices, internal ones and boundary ones. As such, Giraph++ does not support block-level communication, i.e., message passing is still from vertices to vertices, rather than from blocks to blocks which is more efficient for solving many graph problems. For example, an algorithm for computing connected components is used to demonstrate Giraph++ programming in [49], in which vertex IDs are passed among internal vertices and boundary vertices. We will implement an algorithm for the same problem in our block-centric framework in Section 3.2, and we will show that it is much simpler and more efficient to simply exchange block IDs between blocks directly. Moreover, since Giraph++ is Java-based, the intra-block computation inevitably incurs (de)serialization cost for each vertex accessed; in contrast, the intra-block computation in Blogel is simply a main-memory algorithm without any additional cost. Finally, Giraph++ extends METIS [24] for graph partitioning, which is an expensive method. In Section 3.7, we will show that graph partitioning and computing is much more efficient in Blogel than in Giraph++.

Another recent system, GRACE [52], which works in a single-machine environment, also applies graph partitioning to improve performance. Instead of allowing block-centric computing, GRACE enhances vertex-centric computing with a scheduler that controls the order of vertex computation in a block. This can be regarded as a special case of block-centric computing, where the intra-block computing logic is totally defined by the scheduler. Although this model relieves users of the burden of implementing the intra-block computing logic, it is not as expressive as our block-centric paradigm. For example, since GRACE does not allow a block to have its own states and does not support block-level communication, it has the same limitation as Giraph++'s computing framework. We also remark that Blogel and GRACE have different focuses: GRACE attempts to improve main memory bandwidth utilization in a single-machine environment, while Blogel aims to reduce the computation and communication workload in a distributed environment.

3.1.2 GraphLab and PowerGraph

Unlike Pregel's synchronous model and message passing paradigm, GraphLab adopts a shared memory abstraction and supports asynchronous execution. GraphLab is first developed as a parallel graph computing abstraction on a single machine [29], and is extended to work in distributed environments later in [30]. It hides the communication details from programmers: when processing a vertex, users can directly access its own field as well as the fields of its adjacent edges and vertices. GraphLab maintains a global scheduler, and workers fetch vertices from the scheduler for processing, possibly adding the neighbors of these vertices into the scheduler. Asynchronous computing may decrease the workload for some algorithms but incurs extra cost due to data locking/unlocking.

To address the imbalanced workload caused by high-degree vertices, a recent version of GraphLab, called PowerGraph [17], partitions the graph by edges instead of by vertices, so that the edges of a high-degree vertex are handled by multiple workers. Accordingly, an edge-centric Gather-Apply-Scatter (GAS) computing model is used. However, GraphLab is less expressive than Pregel, since a vertex cannot communicate with a non-neighbor, and graph mutation is not supported.

3.2 Merits of Block-Centric Computing: a First Example

We first use an example to illustrate the main differences in the programming and performance between the block-centric model and vertex-centric model, by considering the

		Computing Time	Total Msg #	Superstep #
BTC	**Vertex-Centric**	28.48 s	1,188,832,712	30
	Block-Centric	0.94 s	1,747,653	6
Friendster	**Vertex-Centric**	120.24 s	7,226,963,186	22
	Block-Centric	2.52 s	19,410,865	5
USA Road	**Vertex-Centric**	510.98s	8,353,044,435	6,262
	Block-Centric	1.94 s	270,257	26

Figure 3.1: Overall Performance of Hash-Min

Superstep #		1	2	3	4	5	6	7	8	9		29	30
Vertex-Centric	**Time**	7.24 s	6.82 s	6.62 s	2.86 s	2.34 s	0.17 s	0.13 s	0.10 s	0.09s	…	0.08 s	0.10 s
	Msg #	393,175,048	349,359,723	320,249,398	78,466,694	44,961,718	1,884,460	530,278	128,602	61,727		4	0
Block-Centric	**Time**	0.29 s	0.12 s	0.10 s	0.15 s	0.12 s	0.15 s						
	Msg #	1,394,408	294,582	55,775	2,848	40	0						

(a) Per-Superstep Performance over BTC

Superstep #		1	2	3	4	5	6	7	8		21	22
Vertex-Centric	**Time**	26.86 s	27.64 s	27.86 s	26.97 s	8.96 s	0.43 s	0.15 s	0.11 s	…	0.18 s	0.15 s
	Msg #	1,725,523,081	1,719,438,065	1,717,496,808	1,636,980,454	416,289,356	8,780,258	1,484,531	587,275		1	0
Block-Centric	**Time**	0.53 s	1.53 s	0.25 s	0.10 s	0.06 s						
	Msg #	6,893,957	6,892,723	5,620,051	4,134	0						

(b) Per-Superstep Performance over Friendster

Figure 3.2: Performance of Hash-Min Per Superstep

Hash-Min algorithm of [37] for finding connected components (CCs). We will show the merits of the block-centric model for processing large real-world graphs with the three characteristics discussed in Section 1.2.

Given a CC C, we denote the set of vertices of C by $V(C)$, and define the ID of a CC C to be $cc(v) = \min\{id(u) : u \in V(C)\}$. Hash-Min computes $cc(v)$ for each vertex $v \in V$. The idea is to broadcast the smallest vertex ID seen so far by each vertex v, denoted by $min(v)$.

For the vertex-centric model, in superstep 1, each vertex v sets $min(v)$ to be the smallest ID among $id(v)$ and $id(u)$ of all $u \in \Gamma(v)$, broadcasts $min(v)$ to all its neighbors, and votes to halt. In each later superstep, each vertex v receives messages from its neighbors; let min^* be the smallest ID received, if $min^* < min(v)$, v sets $min(v) = min^*$ and broadcasts min^* to its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, $min(v) = cc(v)$ for all v.

Next we discuss the implementation of Hash-Min in the block-centric framework. Let us assume that vertices are already grouped into blocks by our partitioners (to be discussed in Section 3.6), which guarantee that vertices in a block are connected. Each vertex belongs to a unique block, and let $block(v)$ be the ID of the block that v belongs to. Let $\mathbb{B}$

be the set of blocks, and for each block $B \in \mathbb{B}$, let $V(B)$ be the set of vertices of B, and $id(B)$ be the integer ID of B. We define $\Gamma(B)=\bigcup_{v \in V(B)}\{block(u) : u \in \Gamma(v)\}$. Thus, we obtain a *block-level graph*, $\mathbb{G} = (\mathbb{B}, \mathbb{E})$, where $\mathbb{E} = \{(B_i, B_j) : B_i \in \mathbb{B}, B_j \in \Gamma(B_i)\}$. We then simply run Hash-Min on $\mathbb{G}$ where blocks in $\mathbb{B}$ broadcast the smallest block ID that they have seen. Similar to the vertex-centric algorithm, each block B maintains a field $min(B)$, and when the algorithm converges, all vertices v with the same $min(block(v))$ belong to one CC.

We now analyze why the block-centric model is more appealing than the vertex-centric model. First, we consider the processing of graphs with skewed degree distribution, where we compare G with $\mathbb{G}$. Most real-world graphs with skewed degree distribution consists of a giant CC and many small CCs. In this case, our partitioner computes a set of roughly even-sized blocks from the giant CC, while each small CC forms a block. Let b be the average number of vertices in a block and d_{max} be the maximum vertex degree in G. The vertex-centric model works on G and hence a high-degree vertex may send/receive $O(d_{max})$ messages each round, causing skewed workload among the workers. On the contrary, the block-centric model works on $\mathbb{G}$ and a high-degree vertex involves at most $O(n/b)$ messages each round, where n is the number of vertices in the giant CC. For power-law graphs, d_{max} can approach n and n/b can be a few orders of magnitude smaller than d_{max} for a reasonable setting of b (e.g., $b = 1000$).

In addition, as long as the number of blocks is sufficiently larger than the number of workers, the block-centric model can achieve a rather balanced workload with a simple greedy algorithm for block-to-worker assignment described in Section 3.3.

Figure 3.1 reports the overall performance (elapsed computing time, total number of messages sent, and total number of supersteps) of the vertex-centric and block-centric Hash-Min algorithms on three graphs: an RDF graph *BTC*, a social network *Friendster*, and a *USA* road network. The details of these graphs can be found in Section 3.7. Figure 3.2 reports the performance (running time and number of messages) of Hash-Min for each superstep on *BTC* and *Friendster*.

BTC has skewed degree distribution, as a few vertices have degree over a million but the average degree is only 4.69. The vertex-centric program is severely affected by the skewed workload due to high-degree vertices in the first few supersteps, and a large number of messages are passed as shown in Figure 3.2(a). The number of messages starts to drop significantly only in superstep 4 when the few extremely high-degree vertices become inactive. The subsequent supersteps involve less and less messages since the smallest vertex IDs have been seen by most vertices, and hence only a small fraction of the vertices remain active. However, it still takes 30 supersteps to complete the job. On the contrary, our block-centric program has balanced workload from the beginning and uses only 6 supersteps. Overall, as Figure 3.1 shows, the block-centric program uses 680 times less messages and 30 times less computing time.

Next, we discuss the processing of graphs with high density. Social networks and mobile phone networks often have relatively higher average degree than other real-world graphs. For example, Friendster has average degree of 55.06 while the USA road network has average degree of only 2.44 (see Figure 3.7), which implies that a vertex of Friendster can send/receive 22.57 times more messages than that of the USA road network. The total number of messages in each superstep of the vertex-centric Hash-Min is bounded by $O(|E|)$, while that of the block-centric Hash-Min is bounded by $O(|\mathbb{E}|)$. Since $|\mathbb{E}|$ is generally significantly smaller than $|E|$, the block-centric model is much cheaper.

As Figure 3.1 shows, the vertex-centric Hash-Min uses 372 times more messages than the block-centric Hash-Min on Friendster, resulting in 48 times longer computing time. Figure 3.2(b) further shows that the number of messages and the elapsed computing time per superstep of the vertex-centric program is significantly larger than that of the block-centric program. Note that the more message passing and longer computing time are not primarily due to skewed workload because the maximum degree in Friendster is only 5214.

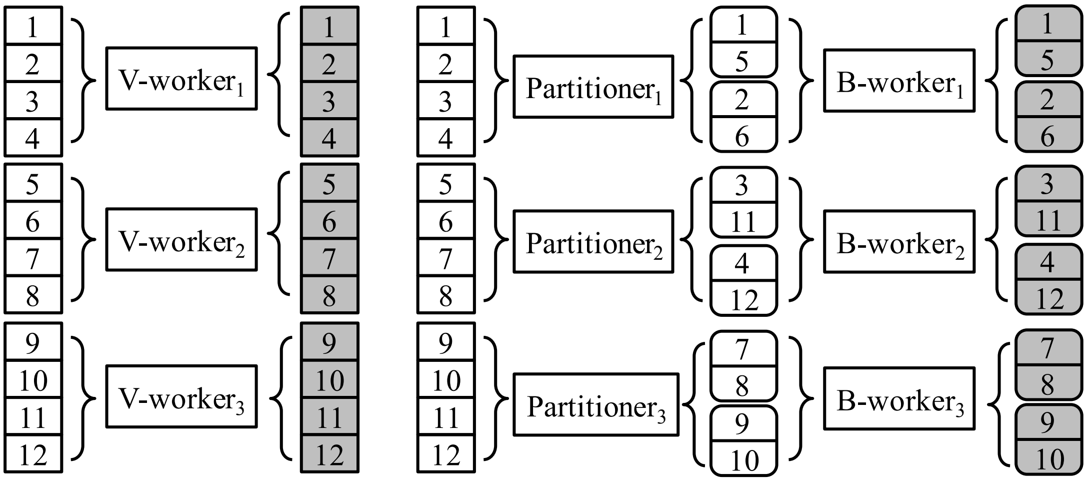
Let $\delta(G)$ and $\delta(\mathbb{G})$ be the diameter of G and $\mathbb{G}$. The vertex-centric Hash-Min takes $O(\delta(G))$ supersteps, while the block-centric Hash-Min takes only $O(\delta(\mathbb{G}))$ supersteps. For real-world graphs with a large diameter, such as road networks, condensing a large region (i.e., a block) of G into a single vertex in $\mathbb{G}$ allows distant points to be reached within a much smaller number of hops. For example, in the USA road network, there may be thousands of hops between Washington and Florida; but if we condense each state into a block, then the two states are just a few hops apart.

The adverse effect of large diameter on the vertex-centric model can be clearly demonstrated by the USA road network. As shown in Figure 3.1, the vertex-centric Hash-Min takes in 6,262 supersteps, while the block-centric Hash-Min runs on a block-level graph with a much small diameter and uses only 22 supersteps. The huge number of supersteps of the vertex-centric Hash-Min also results in 263 times longer computing time and 30,908 times more messages being passed than the block-centric algorithm.

3.3 System Overview

We first give an overview of the Blogel framework. Blogel supports three types of jobs: (1)*vertex-centric graph computing*, where a worker is called a *V-worker*; (2)*graph partitioning* which groups vertices into blocks, where a worker is called a *partitioner*; (3)*block-centric graph computing*, where a worker is called a *B-worker*.

Figure 3.3 illustrates how the three types of workers operate. Figure 3.3(a) shows that each V-worker loads its own portion of vertices, performs vertex-centric graph compu-



(a) V-workers (b) Partitioners and B-workers

Figure 3.3: Operating Logic of Different Types of Workers

tation, and then dumps the output of the processed vertices (marked as grey) to HDFS. Figure 3.3(b) shows block-centric graph computing, where we assume that every block contains two vertices. Specifically, the vertices are first grouped into blocks by the partitioners, which dump the constructed blocks to HDFS. These blocks are then loaded by the B-workers for block-centric computation.

Blocks and B-workers. In both vertex-centric graph computing and graph partitioning, when a vertex v sends a message to a neighbor $u \in \Gamma(v)$, the worker to which the message is to be sent is identified by hashing $id(u)$. We now consider block-centric graph computing. Suppose that the vertex-to-block assignment and block-to-worker assignment are already given, we define the block of a vertex v by $block(v)$ and the worker of a block B by $worker(B)$. We also define $worker(v) = worker(block(v))$. Then, the ID of a vertex v in a B-worker is now given by a triplet $trip(v) = \langle id(v), block(v), worker(v)\rangle$. Thus, a B-worker can obtain the worker and block of a vertex v by checking its ID $trip(v)$.

Similar to a vertex in Pregel, a block in Blogel also has a *compute*() function. We use *B-compute*() and *V-compute*() to denote the *compute*() function of a block and a vertex, respectively. A block has access to all its vertices, and can send messages to any block B or vertex v as long as $worker(B)$ or $worker(v)$ is available. Each B-worker maintains two message buffers, one for exchanging vertex-level messages and the other for exchanging block-level messages. A block also has a state indicating whether it is active, and may vote to halt.

Blogel computing modes. Blogel operates in one of the following three computing modes, depending on the application:

- **B-mode.** In this mode, Blogel only calls *B-compute*() for all its blocks, and only block-level message exchanges are allowed. A job terminates when all blocks voted to halt and there is no pending message for the next superstep.
- **V-mode.** In this mode, Blogel only calls *V-compute*() for all its vertices, and only vertex-level message exchanges are allowed. A job terminates when all vertices voted to halt and there is no pending message for the next superstep.
- **VB-mode.** In this mode, in each superstep, a B-worker first calls *V-compute*() for all its vertices, and then calls *B-compute*() for all its blocks. If a vertex v receives a message at the beginning of a superstep, v is activated along with its block $B = block(v)$, and B will call its *B-compute*() function. A job terminates only if all vertices and blocks voted to halt and there is no pending message for the next superstep.

Partitioners. Blogel supports several types of pre-defined partitioners. Users may also implement their own partitioners in Blogel. A partitioner loads each vertex v together with $\Gamma(v)$. If the partitioning algorithm supports only undirected graphs but the input graph G is directed, partitioners first transform G into an undirected graph, by making each edge bi-directed. The partitioners then compute the vertex-to-block assignment (details in Section 3.6).

Block assignment. After graph partitioning, $block(v)$ is computed for each vertex v. The partitioners then compute the block-to-worker assignment. This is actually a load balancing problem:

Definition 1 (Load Balancing Problem [26]). Given k workers $w_1, \ldots, w_k$, and n jobs, let $J(i)$ be the set of jobs assigned to worker w_i and c_j be the cost of each job j. The load of worker w_i is defined as $L_i = \sum_{j \in J(i)} c_j$. The goal is to minimize $L = \max_i L_i$.

In our setting, each job corresponds to a block B, whose cost is given by the number of vertices in B. We use the 4/3-approximation algorithm given in [16], which first sorts the jobs by cost in non-increasing order, and then scans through the sorted jobs and assigns each job to the worker with the smallest load.

The block-to-worker assignment is computed as follows. (1)To get the block sizes, each partitioner groups its vertices into blocks and sends the number of vertices in each block to the master. The master then aggregates the numbers sent by the partitioners to obtain the global number of vertices for each block. (2)The master then computes the block-to-worker assignment using the greedy algorithm described earlier, and broadcasts the assignment to each partitioner. (3)Each partitioner sets $worker(v)$ for each of its vertex v according to the received block-to-worker assignment (note that $block(v)$ is already computed).

Triplet ID of neighbors. So far we only compute $trip(v)$ for each vertex v. However, in block-centric computing, if a vertex v needs to send a message to a neighbor $u \in \Gamma(v)$, it needs to first obtain the worker holding u from $trip(u)$. Thus, we also compute $\widehat{\Gamma}(v) = \{trip(u) : u \in \Gamma(v)\}$ for each vertex v as follows. Each worker w_i constructs a look-up table $LT_{i\to j}$ locally for every worker w_j in the cluster: for each vertex v in w_i, and for each neighbor $u \in \Gamma(v)$, $trip(v)$ is added to $LT_{i\to j}$, where $j = hash(id(u))$, i.e., u is on worker w_j. Then, w_i sends $LT_{i\to j}$ to each w_j, and each worker merges the received look-up tables into one look-up table. Now, a vertex u on worker w_j can find $trip(v)$ for each neighbor $v \in \Gamma(u)$ from the look-up table of w_j to construct $\widehat{\Gamma}(u)$.

Till now, each partitioner w_i still holds only those vertices v with $hash(id(v)) = i$. After $\widehat{\Gamma}(v)$ is constructed for each vertex v, the partitioners exchange vertices according to the block-to-worker assignment. Each partitioner w_i then dumps its vertices to an HDFS file, which is later loaded by the corresponding B-worker w_i during block-centric computing. Each vertex v has an extra field, $content(v)$, that keeps additional information such as edge weight and edge direction during data loading. It is used along with $trip(v)$ and $\widehat{\Gamma}(v)$, to format v's output line during data dumping.

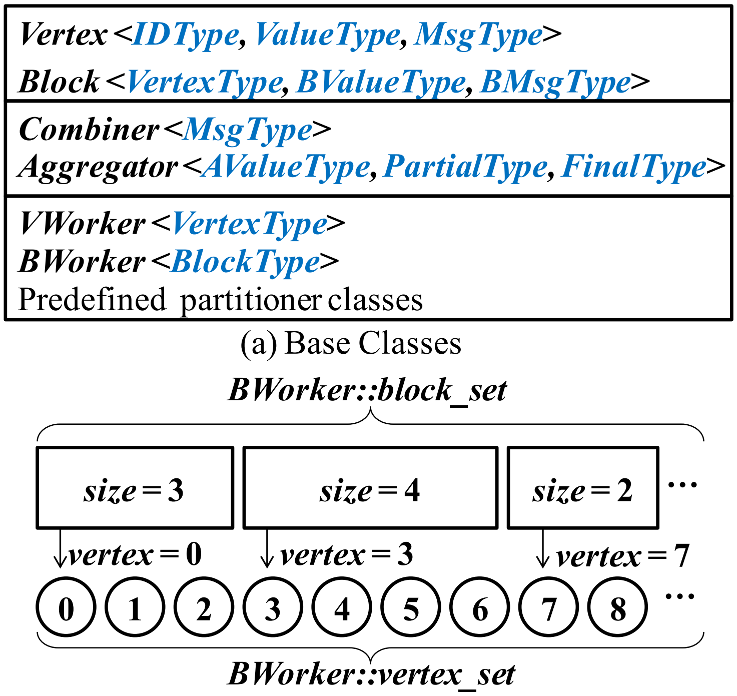
3.4 Programming Interface

Similar to Pregel, writing a Blogel program involves subclassing a group of predefined classes, with the template arguments of each base class properly specified. Figure 3.4(a) shows the base classes of Blogel. We illustrate their usage by showing how to implement the *Hash-Min* algorithm described in Section 3.2.

3.4.1 Vertex-Centric Interface

In the vertex-centric model, each vertex v maintains an integer ID $id(v)$, an integer field $cc(v)$ and a list of neighbors $\Gamma(v)$. Thus, in the *Vertex* class, we specify *IDType* as integer, *ValueType* as a user-defined type for holding both $cc(v)$ and $\Gamma(v)$, and *MsgType* as integer since *Hash-Min* sends messages in the form of vertex ID. We then define a class *CCVertex* to inherit this class, and implement the *compute*() function using the *Hash-Min* algorithm in Section 3.2. The *Vertex* class also has another template argument (not shown in Figure 3.4) for specifying the vertex-to-worker assignment function over *IDType*, for which a hash function is specified by default.

To run the *Hash-Min* job, we inherit the *VWorker<CCVertex>* class, and implement two functions: (1)*load_vertex*(*line*), which specifies how to parse a line from the input file into a vertex object; (2)*dump_vertex*(*vertex*, *HDFSWriter*), which specifies how to dump a vertex object to the output file. *VWorker*'s *run*() function is then called to start the job.



(a) Base Classes

(b) Vertex Sets and Block Sets

Figure 3.4: Programming Interface of Blogel

BWorker and the partitioner classes also have these three functions, though the logic of *run*() is different.

3.4.2 Block-Centric Interface

For the block-centric model, in the *Vertex* class, we specify *IDType* as a triplet for holding $trip(v)$, *ValueType* as a list for holding $\widehat{\Gamma}(v)$, while *MsgType* can be any type since the algorithm works in B-mode and there is no vertex-level message exchange. We then define a class *CCVertex* that inherits this class with an empty *V-compute*() function. In the *Block* class, we specify *VertexType* as *CCVertex*, *BValueType* as a block-level adjacency list (i.e., $\Gamma(B)$), and *BMsgType* as integer since the algorithm sends messages in the form of block ID. We then define a class *CCBlock* that inherits this *Block* class, and implement the *B-compute*() function using the logic of block-centric *Hash-Min*. Finally, we inherits the *BWorker*<*CCBlock*> class, implement the vertex loading/dumping functions, and call *run*() to start the job.

A *Block* object also has the following fields: (1)an integer block ID, (2)an array of the block's vertices, denoted by $vertex$, (3)and the number of vertices in the block, denoted by $size$.

A *BWorker* object also contains an array of blocks, denoted by $block\_set$, and an array of vertices, denoted by $vertex\_set$. As shown in Figure 3.4(b), the vertices in a B-worker's $vertex\_set$ are grouped by blocks; and for each block B in the B-worker's $block\_set$, B's $vertex$ field is actually a pointer to the first vertex of B's group in the B-worker's $vertex\_set$. Thus, a block can access its vertices in *B-compute*() as $vertex[0]$, $vertex[1]$, $\ldots$, $vertex[size-1]$.

A subclass of *BWorker* also needs to implement an additional function *block_init*(), which specifies how to set the user-specified field *BValueType* for each block. After a B-worker loads all its vertices into *vertex_set*, it scans *vertex_set* to construct its *block_set*, where the fields *vertex* and *size* of each block are automatically set. Then, before the block-centric computing begins, *block_init*() is called to set the user-specified block field. For our block-centric *Hash-Min* algorithm, in *block_init*(), each block B constructs $\Gamma(B)$ from $\Gamma(v)$ of all $v \in V(B)$.

3.4.3 Global Interface

In Blogel, each worker is a computer process. We now look at a number of fields and functions that are global to each process. They can be accessed in both *V-compute*() and *B-compute*().

Message buffers and combiners. Each worker maintains two message buffers, one for exchanging vertex-level messages and the other for exchanging block-level messages. A combiner is associated with each message buffer, and it is not defined by default. To use a combiner, we inherit the *Combiner* class and implement the *combine*() function, which specifies the combiner logic. When a vertex or a block calls the *send_msg*(*target*, *msg*) function of the message buffer, *combine*() is called if the combiner is defined.

Aggregator. An aggregator works as follows. In each superstep, a vertex/block's *V-compute*()/*B-compute*() function may call *aggregate*(*value*), where *value* is of type *AValueType*. After a worker calls *V-compute*()/*B-compute*() for all vertices/blocks, the aggregated object maintained by its local aggregator (of type *PartialType*) is sent to the master. When the master obtains the locally aggregated objects from all workers, it calls *master_compute*() to aggregate them to a global object of type *FinalType*. This global object is then broadcast to all the workers so that it is available to every vertex/block in the next superstep.

To define an aggregator, we subclass the *Aggregator* class to include a field recording the aggregated values, denoted by agg, and implement *aggregate*(*value*) and *master_compute*(). An object of this class is then assigned to the worker.

Blogel also supports other useful global functions such as graph mutation (which are user-defined functions to add/delete vertices/edges), and the *terminate*() function which can be called to terminate the job immediately. In addition, Blogel maintains the following global fields which are useful for implementing the computing logic: (1)the ID of the current superstep, which also indicates the number of supersteps executed so far; (2)the total number of vertices among all workers at the beginning of the current superstep, whose value may change due to graph mutation; (3)the total number of active vertices among all workers at the beginning of the current superstep.

3.5 Applications

We apply Blogel to solve four classic graph problems: *Connected Components (CCs)*, *Single-Source Shortest Path (SSSP)*, *Reachability*, and *PageRank*. In Sections 3.2 and 3.4, we have discussed how Blogel computes CCs with the *Hash-Min* logic in both the vertex-centric model, and B-mode of the block-centric model. We now discuss the solutions to the other three problems in Blogel.

3.5.1 Single-Source Shortest Path

Given a graph $G = (V, E)$, where each edge $(u, v) \in E$ has length $\ell(u, v)$, and a source $s \in V$, SSSP computes a shortest path from s to every other vertex $v \in V$, denoted by $SP(s, v)$.

Vertex-centric algorithm. We first discuss the vertex-centric algorithm, which is similar to Pregel's SSSP algorithm [33]. Each vertex v has two fields: $\langle prev(v), dist(v)\rangle$ and $\Gamma_{out}(v)$, where $prev(v)$ is the vertex preceding v on $SP(s, v)$ and $dist(v)$ is the length of $SP(s, v)$. Each $u \in \Gamma_{out}(v)$ is associated with $\ell(v, u)$.

Initially, only s is active with $dist(s)$=0, while $dist(v)$=∞ for any other vertex v. In superstep 1, s sends a message $\langle s, dist(s) + \ell(s, u)\rangle$ to each $u$$\in$$\Gamma_{out}(s)$, and votes to halt. In superstep i (i>1), if a vertex v receives messages $\langle w, d(w)\rangle$ from any of v's in-neighbor w, then v finds the in-neighbor w^* such that $d(w^*)$ is the smallest among all $d(w)$ received. If $d(w^*) < dist(v)$, v updates $\langle prev(v), dist(v)\rangle$=$\langle w^*, d(w^*)\rangle$, and sends a message $\langle v, dist(v) + \ell(v, u)\rangle$ to each out-neighbor $u \in \Gamma_{out}(v)$. Finally, v votes to halt.

Let $hop(s, v)$ be the number of hops of $SP(s, v)$, and $\mathcal{L} = \max_{v\in V} hop(s, v)$. The vertex-centric algorithm runs for $O(\mathcal{L})$ supersteps and in each superstep, at most one message is sent along each edge. Thus, the total workload is $O(\mathcal{L}(|V| + |E|))$.

Block-centric algorithm. Our block-centric solution operates in VB-mode. Each vertex maintains the same fields as in the vertex-centric algorithm, and blocks do not maintain any information. In each superstep, *V-compute*() is first executed for all vertices, where a vertex v finds w^* from the incoming messages as in the vertex-centric algorithm. However, now v votes to halt only if $d(w^*) \geq dist(v)$. Otherwise, v updates $\langle prev(v), dist(v)\rangle = \langle w^*, d(w^*)\rangle$ but stays active. Then, ***B-compute***() is executed, where each block B collects all its active vertices v into a priority queue Q (with $dist(v)$ as the key), and makes these vertices vote to halt. ***B-compute***() then runs Dijkstra's algorithm on B using Q, which removes the vertex $v \in Q$ with the smallest value of $dist(v)$ from Q for processing each time. The out-neighbors $u \in \Gamma(v)$ are updated as follows. For each $u \in V(B)$, if $dist(v) + \ell(v, u) < dist(u)$, we update $\langle prev(u), dist(u)\rangle$ to be $\langle v, dist(v) + \ell(v, u)\rangle$, and insert u into Q with key $dist(u)$ if $u \notin Q$, or update $dist(u)$ if u is already in Q. For each $u \notin V(B)$, a message $\langle v, dist(v) + \ell(v, u)\rangle$ is sent to u. B votes to halt when Q becomes empty. In the next superstep, if a vertex u receives a message, u is activated along with its block, and the block-centric computation repeats.

Compared with the vertex-centric algorithm, this algorithm saves a significant amount of communication cost since there is no message passing among vertices within each block. In addition, messages propagate from s in the unit of blocks, and thus, the algorithm requires much less than supersteps than $O(\mathcal{L})$.

For both the vertex-centric and block-centric algorithms, we apply a combiner as follows. Given a set of messages from a worker, $\{\langle w_1, d(w_1)\rangle, \langle w_2, d(w_2)\rangle, \ldots, \langle w_k, d(w_k)\rangle\}$, to be sent to a vertex u, the combiner combines them into a single message $\langle w^*, d(w^*)\rangle$ such that $d(w^*)$ is the smallest among all $d(w_i)$ for $1 \leq i \leq k$.

3.5.2 Reachability

Given a directed graph $G = (V, E)$, a source vertex s and a destination vertex t, the problem is to decide whether there is a directed path from s to t in G. We can perform a bidirectional breadth-first search (BFS) from s and t, and check whether the two BFSs meet at some vertex. We assign each vertex v a 2-bit field $tag(v)$, where the first bit indicates whether s can reach v and the second bit indicates whether v can reach t.

Vertex-centric algorithm. We first set $tag(s) = 10$, $tag(t) = 01$; and for any other vertex v, $tag(v) = 00$. Only s and t are active initially. In superstep 1, s sends its tag 10 to all $v \in \Gamma_{out}(s)$, and t sends its tag 01 to all $v \in \Gamma_{in}(t)$. They then vote to halt. In superstep i ($i > 1$), a vertex v computes the bitwise-OR of all messages it receives, which results in tag^*. If $tag^* = 11$, or if the bitwise-OR of tag^* and $tag(v)$ is 11, v sets $tag(v) = 11$ and calls *terminate*() to end the program since the two BFSs now meet at v; otherwise, if $tag(v) = 00$, v sets $tag(v) = tag^*$ and, either sends tag^* to all $u \in \Gamma_{out}(v)$ if $tag^* = 10$, or sends tag^* to all $u \in \Gamma_{in}(v)$ if $tag^* = 01$. Finally, v votes to halt.

Note that if we set t to be a non-existent vertex ID (e.g., -1), the algorithm becomes BFS from s. We now analyze the cost of doing BFS. Let V_h be the set of vertices that are

h hops away from s. We can prove by induction that, in superstep i, only those vertices in V_i both receive messages (from the vertices in V_{i-1}) and send messages to their out-neighbors. If a vertex in V_j ($j < i$) receives a message, it simply votes to halt without sending messages; while all vertices in V_j ($j > i$) remain inactive as they are not reached yet. Thus, the total workload is $O(|E| + |V|)$.

Block-centric algorithm. This algorithm operates in VB-mode. In each superstep, *V-compute*() is first called where each vertex v receives messages and updates $tag(v)$ as in the vertex-centric algorithm. If $tag(v)$ is updated, v remains active; otherwise, v votes to halt. Then, *B-compute*() is called, where each block B collects all its active vertices with tag 10 (01) into a queue Q_s (Q_t). If an active vertex with tag 11 is found, B calls *terminate*(). Otherwise, B performs a forward BFS using Q_s as follows. A vertex v is dequeued from Q_s each time, and the out-neighbors $u \in \Gamma_{out}(v)$ are updated as follows. For each out-neighbor $u \in V(B)$, if $tag(u) = 00$, $tag(u)$ is set to 10 and u is enqueued; if $tag(u) = 01$ or 11, $tag(u)$ is set to 11 and *terminate*() is called. For each out-neighbor $u \notin V(B)$, a message 10 is sent to u. Then a backward BFS using Q_t is performed in a similar way. Finally, B votes to halt. In the next superstep, if a vertex u receives a message, u is activated along with its block, and the block-centric computation repeats.

3.5.3 PageRank

Given a directed graph $G = (V, E)$, the problem is to compute the PageRank of each vertex $v \in V$. Let $pr(v)$ be the PageRank of v. Pregel's PageRank algorithm [33] works as follows. In superstep 1, each vertex v initializes $pr(v) = 1/|V|$ and distributes $pr(v)$ to the out-neighbors by sending each one $pr(v)/|\Gamma_{out}(v)|$. In superstep i ($i > 1$), each vertex v sums up the received PageRank values, denoted by sum, and computes $pr(v) = 0.15/|V| + 0.85 \times sum$. It then distributes $pr(v)/|\Gamma_{out}(v)|$ to each out-neighbor. A combiner is used, which aggregates the messages to be sent to the same destination vertex into a single message that equals their sum.

PageRank loss. Conceptually, the total amount of PageRank values remain to be 1, with 15% held evenly by the vertices, and 85% redistributed among the vertices by propagating along the edges. However, if there exists a sink page v (i.e., $\Gamma_{out}(v) = \emptyset$), $pr(v)$ is not distributed to any other vertex in the next superstep and the value simply gets lost. Therefore, in [33]'s PageRank algorithm, the total amount of PageRank values decreases as the number of supersteps increases.

Let $V_0 = \{v \in V : \Gamma_{out}(v) = \emptyset\}$ be the set of *sink vertices*, i.e., vertices with out-degree 0. We ran [33]'s PageRank algorithm on two web graphs (listed in Figure 3.7) and computed the PageRank loss: (1)*WebUK* which has 9.08% of the vertices being sink vertices, and (2)*WebBase* which has 23.41% of the vertices being sink vertices. We found

	|*V*|	**|*E*|**	**Kendall Tau Distance**
BerkStan	685,230	7,600,595	834,804,094
Google	875,713	5,105,039	2,185,214,827
NotreDame	325,729	1,497,134	1,008,151,095
Stanford	281,903	2,312,497	486,171,631

Figure 3.5: Impact of PageRank Loss

that the algorithm converges in 88 supersteps with 17% PageRank loss on *WebUK*, and in 79 supersteps with 34% PageRank loss on *WebBase*. As we shall see shortly, such a large PageRank loss reveals a problem that must be addressed.

Vertex-centric algorithm. A common solution to the PageRank loss problem is to make each sink vertex $v \in V_0$ link to all the vertices in the graph[1], i.e. to distribute $pr(v)/|V|$ to each vertex. Intuitively, this models the behavior of a random surfer: if the surfer arrives at a sink page, it picks another URL at random and continues surfing again. Since $|V|$ is usually large, $pr(v)/|V|$ is small and the impact of v to the PageRank of other vertices is negligible.

Compared with the standard PageRank definition above, Pregel's PageRank algorithm [33] changes the relative importance order of the vertices. To illustrate, we compute PageRank on the four web graphs from the SNAP database[2] using the algorithm in [33] and the standard PageRank definition, and compare the ordered vertex lists obtained. The results in Figure 3.5 show that the two vertex lists have a large Kendall tau distance. For example, the graph *Google* has over 2 million vertex pairs whose order of PageRank magnitude is reversed from the standard definition.

Obviously, materializing $\Gamma_{out}(v) = V$ for each $v \in V_0$ is unacceptable both in space and in communication cost. We propose an aggregator-based solution. In *compute*(), if v's out-degree is 0, v provides $pr(v)$ to an aggregator that computes $agg = \sum_{v \in V_0} pr(v)$. The PageRank of v is now updated by $pr(v) = 0.15/|V| + 0.85 \times (sum + agg/|V|)$, where sum is compensated with $agg/|V| = \sum_{v \in V_0} \frac{pr(v)}{|V|}$.

Let $pr_i(v)$ be the PageRank of v in superstep i. Then, PageRank computation stops if $|pr_i(v) - pr_{i-1}(v)| < \epsilon/|V|$ for all $v \in V$ (note that the average PageRank is $1/|V|$). We set ϵ to be 0.01 throughout this chapter.

We also implement this stop condition using an aggregator that performs logic AND. In *compute*(), each vertex v provides '*true*' to the aggregator if $|pr_i(v) - pr_{i-1}(v)| < \epsilon/|V|$, and '*false*' otherwise. Moreover, if v finds that the aggregated value of the previous superstep is '*true*', it votes to halt directly without doing PageRank computation.

[1] http://www.google.com/patents/US20080075014

[2] http://snap.stanford.edu/data/

Block-centric algorithm. In a web graph, each vertex is a web page with a URL (e.g., `cs.stanford.edu/admissions`), and we can naturally group all vertices with the same host name (e.g., `cs.stanford.edu`) into a block. Kamvar et al. [20] proposed to initialize the PageRank values by exploiting this block structure, so that PageRank computation can converge faster. Note that though a different initialization is used, the PageRank values will still converge to a unique stable state as guaranteed by the Perron-Frobenius theorem.

The implementation of the algorithm of [20] in the Blogel framework consists of two jobs. The first job operates in B-mode. Before computation, in *block_init*(), each block B first computes the local PageRank of each $v \in V(B)$, denoted by $lpr(v)$, by a single-machine PageRank algorithm with B as input. Block B then constructs $\Gamma(B)$ from $\Gamma(v)$ of all $v \in V(B)$ using [20]'s approach, which assigns a weight to each out-edge. Finally, *B-compute*() computes the BlockRank of each block $B \in \mathbb{B}$, denoted by $br(B)$, on $\mathbb{G} = (\mathbb{B}, \mathbb{E})$ using a logic similar to the vertex-centric PageRank, except that BlockRank is distributed to out-neighbors proportionally to the edge weights. The second job operates in V-mode, which initializes $pr(v) = lpr(v) \cdot br(block(v))$ [20], and performs the standard PageRank computation on G.

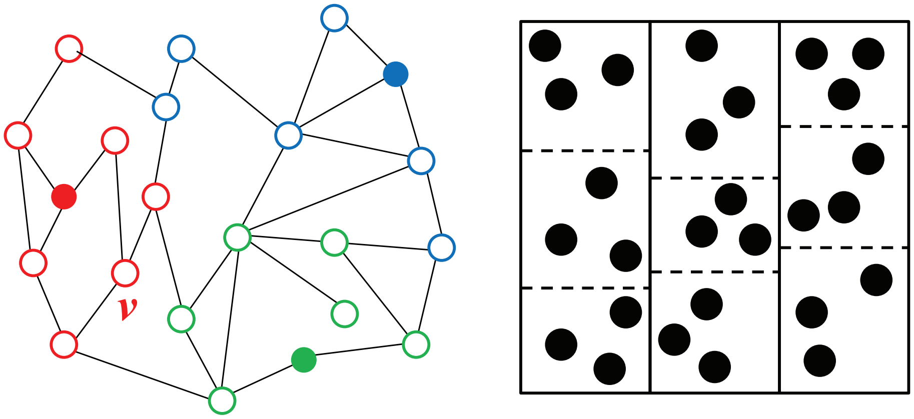
3.6 Partitioners

Efficient computation of blocks that give balanced workload is crucial to the performance of Blogel. We have discussed the logic of partitioners such as the computation of the block-to-worker assignment in Section 3.3. We have also seen a URL-based partitioner for web graphs in Section 3.5.3, where the vertex-to-block assignment is determined by the host names extracted from URLs. In this section, we introduce two other types of partitioners, with an emphasis on computing the vertex-to-block assignment.

3.6.1 Graph Voronoi Diagram Partitioner

We first review the *Graph Voronoi Diagram* (GVD) [13] of an undirected unweighted graph $G = (V, E)$. Given a set of source vertices $s_1, s_2, \ldots, s_k \in V$, we define a partition of V: $\{VC(s_1)$, $VC(s_2)$, ..., $VC(s_k)\}$, where a vertex v is in $VC(s_i)$ only if s_i is closer to v (in terms of the number of hops) than any other source. Ties are broken arbitrarily. The set $VC(s_i)$ is called the Voronoi cell of s_i, and the Voronoi cells of all sources form the GVD of G.

Figure 3.6(a) illustrates the concept of GVD, where source vertices are marked with solid circles. Consider vertex v in Figure 3.6(a), it is at least 2, 3 and 5 hops from the red,



(a) Graph Voronoi Diagram **(b) 2D Partitioner**

Figure 3.6: Partitioners (Best Viewed in Colors)

green and blue sources. Since the red source is closer to v, v is assigned to the Voronoi cell of the red source. All the vertices in Figure 3.6(a) are partitioned into three Voronoi cells, where the vertices with the same color belong to the same Voronoi cell.

The GVD computation can be easily implemented in the vertex-centric computing model, by performing multi-source BFS. Specifically, in superstep 1, each source s sets $block(s) = s$ and broadcasts it to the neighbors; for each non-source vertex v, $block(v)$ is unassigned. Finally, the vertex votes to halt. In superstep i ($i > 1$), if $block(v)$ is unassigned, v sets $block(v)$ to an arbitrary source received, and broadcasts $block(v)$ to its neighbors before voting to halt. Otherwise, v votes to halt directly. When the process converges, we have $block(v) = s_i$ for each $v \in VC(s_i)$.

The multi-source BFS has linear workload since each vertex only broadcasts a message to its neighbors when $block(v)$ is assigned, and thus the total messages exchanged by all vertices is bounded by $O(|E|)$. However, we may obtain some huge Voronoi cells (or blocks), which are undesirable for load balancing. We remark that block sizes can be aggregated at the master in a similar manner as when we compute the block-to-worker assignment in Section 3.3.

Our GVD partitioner works as follows, where we use a parameter $\underline{b_{max}}$ to limit the maximum block size. Initially, each vertex v samples itself as a source with probability $\underline{p_{samp}}$. Then, multi-source BFS is performed to partition the vertices into blocks. If the size of a block is larger than b_{max}, we set $block(v)$ unassigned for any vertex v in the block (and reactivate v). We then perform another round of source sampling and multi-source BFS on those vertices v with $block(v)$ unassigned, using a higher sampling probability. Here, we increase p_{samp} by a factor of $\underline{f}$ ($f \geq 1$) after each round in order to decrease the chance of obtaining an over-sized block. This process is repeated until the stop condition is met.

We check two stop conditions, and the process described above stops as long as one condition is met: (1)let $A(i)$ be the number of active vertices at the beginning of the i-th round of multi-source BFS, then the process stops if $A(i)/A(i-1) > \gamma$. Here, $\underline{\gamma} \leq 1$ is a parameter determining when multi-source BFS is no longer effective in assigning blocks; or (2)the process stops if $p_{samp} > p_{max}$, where $\underline{p_{max}}$ is the maximum allowed sampling rate (recall that $p_{samp} = f \cdot p_{samp}$ after each round).

Moreover, to prevent multi-source BFS from running too many supersteps, which may happen if graph diameter δ is large and the sampling rate p_{samp} is small, we include a user-specified parameter $\underline{\delta_{max}}$ to bound the maximum number of supersteps, i.e., each vertex votes to halt in superstep δ_{max} during multi-source BFS.

When the above process terminates, there may still be some vertices not assigned to any block. This could happen since multi-source BFS works well on large CCs, as a larger CC tends to contain more sampled sources, but the sampling is ineffective for handling small CCs. For example, consider the extreme case where the graph is composed of isolated vertices. Since each round of multi-source BFS assigns block ID to only around $p_{samp}|V|$ vertices, it takes around $1/p_{samp}$ (which is 1000 if $p_{samp} = 0.1\%$) rounds to assign block ID to all vertices, which is inefficient.

Our solution to assigning blocks for small CCs is by the *Hash-Min* algorithm, which marks each small CC as a block using only a small number of supersteps. Specifically, after the rounds of multi-source BFS terminate, if there still exists any unassigned vertex, we run Hash-Min on the subgraph induced by these unassigned vertices. We call this step as *subgraph Hash-Min*.

There are six parameters, $(p_{samp}, \delta_{max}, b_{max}, f, \gamma, p_{max})$, that need to be specified for the partitioner. We show that these parameters are intuitively easy to set. In particular, we found that the following setting of the parameters work well for most large real-world graphs. The sampling rate p_{samp} decides the number of blocks, and usually a small value as 0.1% is a good choice. Note that p_{samp} cannot be too small in order not to create very large blocks. As for the stopping parameters, γ is usually set as 90%, and p_{max} as 10% with $f = 2$, so that there will not be too many rounds of multi-source BFS. The bound on the number of supersteps, δ_{max}, is set to be a tolerable number such as 50, but for small-diameter graphs (e.g., social networks) we can set a smaller δ_{max} such as 10 since the number of supersteps needed for such graphs is small. We set b_{max} to be 100 times that of the expected block size (e.g., $b_{max} = 100,000$ when $p_{samp} = 0.1\%$) for most graphs, except that for spatial networks the block size is limited by δ_{max} already and hence we just set it to ∞. We will further demonstrate in our experiments that the above settings work effectively for different types of real-world graphs.

3.6.2 2D Partitioner

In many spatial networks, vertices are associated with (x, y)-coordinates. Blogel provides a 2D partitioner to partition such graphs. A 2D partitioner associates each vertex v with an additional field (x, y), and it consists of two jobs.

The first job is vertex-centric and works as follows: (1)each worker samples a subset of its vertices with probability p_{samp} and sends the sample to the master; (2)the master first partitions the sampled vertices into n_x slots by the x-coordinates, and then each slot is further partitioned into n_y slots by the y-coordinates. Each resulting slot is a *super-block*. Figure 3.6(b) shows a 2D partitioning with n_x=n_y=3. This partitioning is broadcast to the workers, and each worker assigns each of its vertices to a super-block according to the vertex coordinates. Finally, vertices are exchanged according to the superblock-to-worker assignment computed by master, with $\widehat{\Gamma}(v)$ constructed for each vertex v as described in Section 3.3.

Since a super-block may not be connected, we perform a second block-centric job, where workers run BFS over their super-blocks to break them into connected blocks. Each worker marks the blocks it obtains with IDs 0, 1, $\cdots$. To get the global block ID, each worker w_i sends the number of blocks it has, denoted by $|\mathbb{B}_i|$, to the master which computes for each worker w_j a prefix sum $sum_j = \sum_{i<j} |\mathbb{B}_i|$, and sends sum_j to w_j. Each worker w_j then adds sum_j to the block IDs of its blocks, and hence each block obtains a unique block ID. Finally, $\widehat{\Gamma}(v)$ are updated for each vertex v.

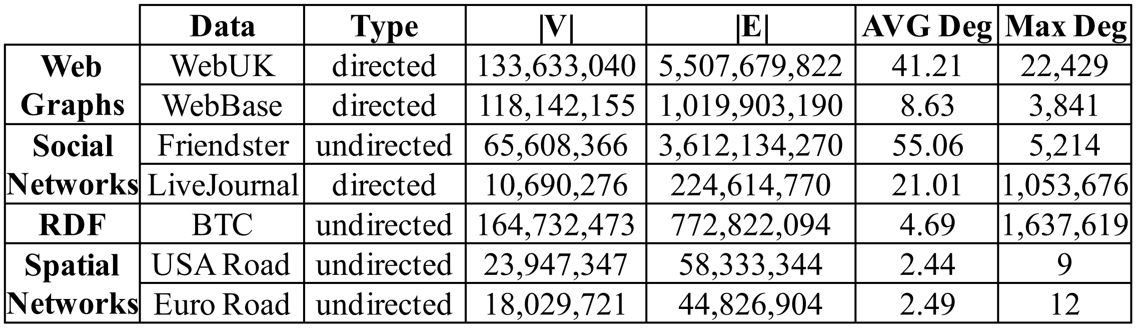
The 2D partitioner has parameters $(p_{samp}$, n_x, $n_y)$. The setting of p_{samp} is similar to the GVD partitioner. To set n_x and n_y, we use $\delta(\mathbb{G}) \approx O(\sqrt{\max\{n_x, n_y\}})$ as a guideline, since the diameter of $\mathbb{G}$ is critical to the performance of block-centric algorithms.

3.7 Experiments

We compare the performance of Blogel and with Giraph 1.0.0, Giraph++[3] and GraphLab 2.2 (which includes the features of PowerGraph [17]). We ran our experiments on a cluster of 16 machines, each with 24 processors (two Intel Xeon E5-2620 CPU) and 48GB RAM. One machine is used as the master that runs only one worker, while the other 15 machines act as slaves running multiple workers. The connectivity between any pair of nodes in the cluster is 1Gbps.

For the experiments of Giraph, we use the multi-threading feature added by Facebook, and thus each worker refers to a computing thread. However, Giraph++ is built on top of an earlier Giraph version by Yahoo!, which does not support multi-threading. We therefore

[3] https://issues.apache.org/jira/browse/GIRAPH-818



	Data	Type	|V|	|E|	AVG Deg	Max Deg
Web Graphs	WebUK	directed	133,633,040	5,507,679,822	41.21	22,429
	WebBase	directed	118,142,155	1,019,903,190	8.63	3,841
Social Networks	Friendster	undirected	65,608,366	3,612,134,270	55.06	5,214
	LiveJournal	directed	10,690,276	224,614,770	21.01	1,053,676
RDF	BTC	undirected	164,732,473	772,822,094	4.69	1,637,619
Spatial Networks	USA Road	undirected	23,947,347	58,333,344	2.44	9
	Euro Road	undirected	18,029,721	44,826,904	2.49	12

Figure 3.7: Datasets

run multiple workers (mapper tasks) per machine. We make all Giraph++ codes used in our experiments public[4].

We used seven large real-world datasets, which are from four different domains as shown in Figure 3.7: (1)web graphs: *WebUK*[5] and *WebBase*[6]; (2)social networks: *Friendster*[7] and *LiveJournal*[8]; (3)RDF graph: *BTC*[9] ; (4)road networks: *USA* and *Euro*[10].

Among them, *WebUK*, *LiveJournal* and *BTC* have skewed degree distribution; *WebUK*, *Friendster* and *LiveJournal* have average degree relatively higher than other large real-world graphs; *USA* and *Euro*, as well as *WebUK*, have a large diameter.

For a graph of certain size, we need a certain amount of computing resources (i.e., workers) to achieve good performance. However, the performance does not further improve if we increase the number of workers per slave machine beyond that amount, since the increased overhead of inter-machine communication outweighs the increased computing power. In the experiments, we run 10 workers for *WebUK* and *WebBase*, 8 for *Friendster*, 2 for *LiveJournal*, 4 for *BTC*, *USA* and *Euro*, which exhibit good performance.

3.7.1 Blogel Implementation

We make Blogel open-source. All the system source codes, as well as the source codes of the applications discussed in this chapter, can be found in `http://www.cse.cuhk.edu.hk/blogel`.

Blogel is implemented in C++ as a group of header files, and users only need to include

[4] https://issues.apache.org/jira/browse/GIRAPH-902

[5] http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05

[6] http://law.di.unimi.it/webdata/webbase-2001

[7] http://snap.stanford.edu/data/com-Friendster.html

[8] http://konect.uni-koblenz.de/networks/livejournal-groupmemberships

[9] http://km.aifb.kit.edu/projects/btc-2009/

[10] http://www.dis.uniroma1.it/challenge9/download.shtml

Data	Parameters						Performance										
							Load	Computation									Dump
								Multi-Source BFS			Subgraph Hash-Min		Block Assign.	Triplet ID Neighbors	Vertex Exchange	Total	
	p_{samp}	δ_{max}	b_{max}	f	γ	p_{max}		Round #	Step #	AVG	Step #	AVG					
WebUK	0.1%	30	500,000	1.6	100%	20%	135.28 s	12	291	0.71 s	8	0.22 s	8.31 s	111.00 s	200.62 s	714.68 s	403.63 s
WebBase	0.1%	30	500,000	2	100%	10%	36.20 s	6	180	0.50 s	70	0.16 s	9.24 s	40.49 s	43.76 s	248.31 s	50.43 s
Friendster	0.1%	10	100,000	2	90%	10%	71.89 s	2	13	4.14 s	12	0.14 s	2.38 s	55.09 s	70.45 s	204.36 s	223.93 s
LiveJournal	0.1%	10	100,000	2	90%	10%	35.08 s	7	64	0.23 s	9	0.20 s	0.87 s	6.14 s	8.56 s	36.19 s	13.71 s
BTC	0.1%	20	500,000	2	95%	10%	29.08 s	3	60	0.60 s	31	0.80 s	2.47 s	12.52 s	25.92 s	112.53 s	39.54 s
USA Road	0.1%	50	∞	2	90%	10%	9.06 s	4	166	0.07 s	19	0.04 s	3.08 s	3.35 s	6.23 s	38.56 s	7.77 s
Euro Road	0.1%	50	∞	2	90%	10%	7.17 s	4	171	0.07 s	17	0.04 s	1.90 s	2.78 s	4.88 s	30.96 s	5.84 s

Figure 3.8: Performance of Graph Voronoi Diagram Partitioners

the necessary base classes and implement the application logic in their subclasses. Blogel communicates with HDFS through libhdfs, a JNI based C API for HDFS. Each worker is simply an MPI process and communications are implemented using MPI communication primitives. While one may deploy Blogel with any Hadoop and MPI version, we use Hadoop 1.2.1 and MPICH 3.0.4 in our experiments. All programs are compiled using GCC 4.4.7 with -O2 option enabled.

Blogel makes the master a worker, and fault recovery can be implemented by a script as follows. The script loops a Blogel job, which runs for at most Δ supersteps before dumping the intermediate results to HDFS. Meanwhile, the script also monitors the cluster condition. If a machine is down, the script kills the current job and restarts another job loading the latest intermediate results from HDFS. Fault tolerance is achieved by the data replication in HDFS.

3.7.2 Performance of Partitioners

We first report the performance of Blogel's partitioners.

Performance of GVD partitioners. Figure 3.8 shows the performance of the GVD partitioners, along with the parameters we used. The web graphs can be partitioned simply based on URLs, but we also apply GVD partitioners to them for the purpose of comparison.

The partitioning parameters are set according to the heuristics given in Section 3.6, but if there exists a giant block in the result which is usually obtained in the phase when Hash-Min is run, we check why the multi-source BFS phase terminates. If it terminates because the sampling rate increases beyond p_{max}, we decrease f and increase p_{max} and γ to allow more rounds of multi-source BFS to be run in order to break the giant block into smaller ones. We also increase b_{max} to relax the maximum block size constraint during multi-source BFS, which may generate some relatively larger blocks (still bounded by b_{max}) but reduce the chance of producing a giant block (not bounded by b_{max}). We slightly adjusted the parameters for *WebUK*, *WebBase* and *BTC* in this way, while all other datasets work well with the default setting described in Section 3.6.

Recall from Section 3.3 that besides block-to-worker assignment, partitioners also compute vertex-to-block assignment (denoted by "Block Assign." in Figure 3.8), construct $\widehat{\Gamma}(v)$ (denoted by "Triplet ID Neighbors"), and exchange vertices according to the block-to-worker assignment (denoted by "Vertex Exchange"). We report the computing time for "Block Assign.", "Triplet ID Neighbors", and "Vertex Exchange" in Figure 3.8. We also report the data loading/dumping time and the total computation time of the GVD partitioners in the gray columns. As for vertex-to-block assignment, (1)for multi-source BFS, we report the number of rounds, the total number of supersteps taken by all these rounds, and the average time of a superstep; (2)for running *Hash-Min* on the subgraphs, we report the number of supersteps and the average time of a superstep.

As Figure 3.8 shows, the partitioning is very efficient as the computing time is comparable to graph loading/dumping time. In fact, within the overall computing time, a significant amount of time is spent on constructing $\widehat{\Gamma}(v)$ and vertex exchange, and the vertex-to-block assignment computation is highly efficient.

Figure 3.9 shows the number of blocks and vertices assigned to each worker, for example, *WebUK* is partitioned over workers 0 to 150 and each worker contains x blocks and y vertices, where $16,781 \leq x \leq 16,791$ and $884,987 \leq y \leq 884,988$. Thus, we can see that the GVD partitioners achieves very balanced block-to-worker assignment, and it also shows that the greedy algorithm described in Section 3.3 is effective. The workload is relatively less balanced only for *BTC*, where we have 13 larger blocks distributed over workers 0 to 12. This is mainly caused by the few vertices in *BTC* with very high degree (the max degree is 1.6M). We remark that probably no general-purpose partitioning algorithm is effective on *BTC*, if there exists a balanced partitioning for *BTC* at all.

Performance of 2D partitioners. Since the vertices in *USA* and *Euro* have 2D coordinates, we also run 2D partitioners on them, with $(p_{samp}, n_x, n_y) = (1\%, 20, 20)$. Figure 3.10(a) shows the quality of 2D partitioning, which reports the number of super-blocks, blocks and vertices in each worker. The number of blocks/vertices per worker obtained by 2D partitioning is not as even as that by GVD partitioning. However, the shape of the super-blocks are very regular, resulting in fewer number of edges crossing super-blocks and a smaller diameter for the block-level graph, and thus block-centric algorithms can run faster.

Figures 3.10(b) and 3.10(c) further show that 2D partitioning is also more efficient than GVD partitioning. For example, for *USA*, job 1 of 2D partitioning takes 26.92 seconds (58% spent on loading and dumping) and job 2 takes 13.29 seconds (93.5% spent on loading and dumping), which is still much shorter than the time taken by the GVD partitioner ($9.06 + 38.56 + 7.77 = 55.39$ seconds).

Comparison with existing partitioning methods. One of the widely used graph parti-

	Worker	Block #	Vertex #
WebUK	0 – 150	16,781 – 16,791	884,987 – 884,988
WebBase	0 – 150	19,329 – 19,341	782,398 – 782,399
Friendster	0 – 120	614 – 618	542,217 – 542,218
LiveJournal	0	5,066	344,848
	1 – 30	5,181 – 5,182	344,847 – 344,848
BTC	0	1	2,673,201
	1 – 12	1	1,337,773 – 1,588,121
	13 – 120	5,902 – 5,928	1,337,679 – 1,337,680
USA Road	0 – 60	4,762 – 4,763	392,579 – 392,580
Eagle Peak	0 – 60	3,564 – 3,569	295,569 – 295,570

Figure 3.9: # of Blocks/Vertices Per Worker (GVD Partitioner)

	Super-Block #	Block #	Vertex #
USA	6 – 7	247 – 606	374,440 – 409,255
Euro	6 – 7	466 – 872	283,695 – 304,775

(a) Per-Worker Statistics

	Load	Partitioning					Dump
		Compute Slots	Block Assign.	Triplet ID Neighbors	Vertex Exchange	Total	
USA	7.49 s	1.75 s	0.03 s	3.70 s	5.82 s	11.31 s	8.12 s
Euro	6.02 s	1.43 s	0.03 s	3.21 s	4.73 s	9.40 s	5.69 s

(b) Job 1 Performance

	Load	Partitioning			Dump
		Compute Blocks	Triplet ID Neighbors	Total	
USA	1.37 s	0.50 s	0.37 s	0.87 s	11.05 s
Euro	1.08 s	0.30 s	0.38 s	0.69 s	5.98 s

(c) Job 2 Performance

Figure 3.10: Performance of 2D Partitioners

tioning algorithms is METIS [24] (e.g., GRACE [52] uses METIS to partition the input graph). However, METIS ran out of memory on those large graphs in Figure 3.7 on our platform. To solve the scalability problem of METIS, Giraph++ [49] proposed a graph coarsening method to reduce the size of the input graph so that METIS can run on the smaller graph. Here, a vertex in the coarsened graph corresponds to a set of connected vertices in the original graph. The partitioning algorithm of Giraph++ consists of 4 phases: (1)graph coarsening, (2)graph partitioning (using METIS), (3)graph uncoarsening, which projects the block information back to the original graph, and (4)ID recoding, which relabels the vertex IDs so that the worker of a vertex v can be obtained by hashing v's new ID. Note that ID recoding is not required in Blogel since the worker ID of each vertex v is stored in its triplet ID $trip(v)$. This approach retains the original vertex IDs so that Blogel's graph computing results require no ID re-projection. Blogel's graph partitioning is also more user-friendly, since it requires only one partitioner job, while Giraph++'s graph partitioning consists of a sequence of over 10 Giraph/MapReduce/METIS jobs.

Figure 3.11 shows the partitioning performance of Giraph++, where we ran as many workers per machine as possible (without running out of memory). We did not obtain

	WebBase	**LiveJournal**	**BTC**	**USA**	**Euro**
Coarsen	3547 s	1234 s	4463 s	228 s	184 s
Partition	395 s	836 s	2360 s	8 s	5 s
Uncoarsen	1414 s	205 s	1064 s	171 s	156 s
IdRecode	94 s	24 s	107 s	25 s	24 s
Total	5450 s	2299 s	7994 s	432 s	369 s

Figure 3.11: Partitioning Performance of Giraph++

	WebUK	**WebBase**	**Friendster**	**LiveJournal**	**BTC**	**USA**	**Euro**
Runtime	4863 s	1373 s	3547 s	205 s	1394 s	127 s	92s

Figure 3.12: Partitioning Performance of LDG

result for *WebUK* and *Friendster*, since graph coarsening ran out of memory even when each slave machine runs only one worker. Figure 3.11 shows that the partitioning time of Giraph++ is much longer than that of our GVD partitioner. For example, while our GVD partitioner partitions *WebBase* in 334.94 seconds (see the breakdown time in Figure 3.8), Giraph++ uses 5450 seconds. In general, our GVD partitioner is tens of times faster than Giraph++'s METIS partitioning algorithm.

Recently, Stanton and Kliot [45] proposed a group of algorithms to partition large graphs, and the best one is a semi-streaming algorithm called Linear (Weighted) Deterministic Greedy (*LDG*). We also ran LDG and the results are presented in Figure 3.12. We can see that LDG is many times slower than our GVD partitioner (reported in Figure 3.8), though LDG is much faster than Giraph++'s new METIS partitioning algorithm.

3.7.3 Partitioner Scalability

We now study the scalability of our GVD partitioner. We first test the partitioning scalability using two real graphs: *BTC* with skewed degree distribution, and *USA* with a large graph diameter. We partition both graphs using varying number of slave machines (i.e., 6, 9, 12 and 15), and study how the partitioning performance scales with the amount of computing resources. We report the results in Figure 3.13. For the larger graph *BTC*, we need a certain amount of computing resources to achieve good performance. For example, when the number of slave machines increases from 6 to 9, the partitioning time of *BTC* improves by 25.7%, from 189.2 seconds to 140.52 seconds. However, the performance does not further improve if we increase the number of machines beyond 12. This is because the increased overhead of inter-machine communication outweighs the increased computing power. For the relatively smaller graph *USA*, the performance does not change much with

		Machine #			
		6	**9**	**12**	**15**
BTC	**Load**	60.35 s	38.50 s	32.00 s	28.71 s
	Compute	189.20 s	140.52 s	119.53 s	115.52 s
	Dump	59.13 s	52.96 s	49.65 s	53.66 s
	Total	308.69 s	231.98 s	201.18 s	197.90 s
USA	**Load**	15.40 s	13.01 s	10.37 s	6.57 s
	Compute	40.05 s	41.32 s	40.40 s	44.74 s
	Dump	6.46 s	4.66 s	4.46 s	3.95 s
	Total	61.91 s	58.99 s	55.23 s	55.26 s

Figure 3.13: GVD Partitioner Scalability on Real Graphs

	|*V*| (avg deg = 20)			
	25M	**50M**	**75M**	**100M**
Load	11.44 s	29.05 s	42.10 s	74.31 s
Compute	45.32 s	88.76 s	127.32 s	164.13 s
Dump	22.26 s	41.64 s	58.86 s	76.84 s
Total	79.02 s	159.45 s	228.29 s	315.28 s

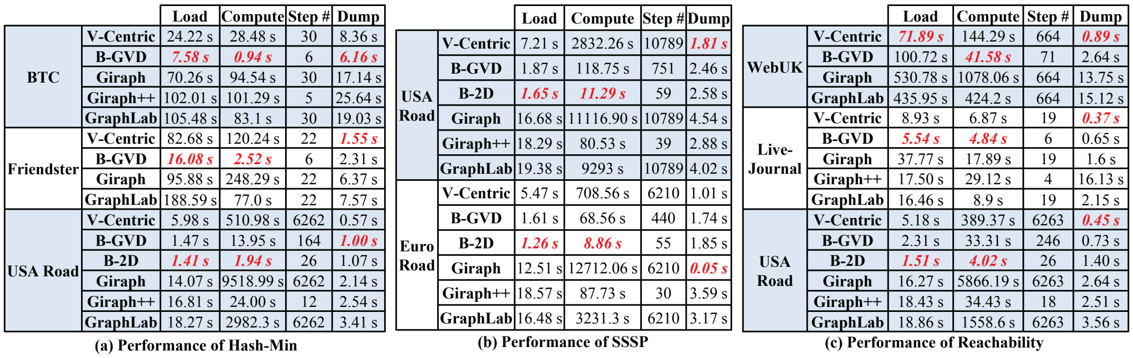
Figure 3.14: GVD Partitioner Scalability on Random Graphs

varying number of slave machines, since the computing power is sufficient even with only 6 slaves.

To test the scalability of our GVD partitioner as the graph size increases, we generate random graphs using PreZER algorithm [34]. We set the average degree as 20 and vary $|V|$ to be 25M, 50M, 75M and 100M. Figure 3.14 shows the scalability results, where all the 16 machines in our cluster are used. The partitioning time increases almost linearly with $|V|$, which verifies that our GVD partitioner scales well with graph size. Moreover, even for a graph with $|V| = 100M$ (i.e., $|E| \approx 2B$), the partitioning is done in only 164.13 seconds. In contrast, even for the smallest graph with $|V| = 25M$, Giraph++'s new METIS partitioning algorithm cannot finish in 24 hours.

3.7.4 Performance of Graph Computing

We now report the performance of various graph computing systems for computing CC, SSSP, reachability, and PageRank. We run the vertex-centric algorithms of Blogel (denoted by *V-centric*), as well as the block-centric algorithm of Blogel (denoted by *B-GVD*, *B-2D* or *B-URL* depending on which partitioner is used). Note that *B-2D* applies to road networks only, *B-URL* applies to web graphs only, while *B-GVD* applies to gen-



		Load	Compute	Step #	Dump
BTC	V-Centric	24.22 s	28.48 s	30	8.36 s
	B-GVD	***7.58 s***	***0.94 s***	6	***6.16 s***
	Giraph	70.26 s	94.54 s	30	17.14 s
	Giraph++	102.01 s	101.29 s	5	25.64 s
	GraphLab	105.48 s	83.1 s	30	19.03 s
Friendster	V-Centric	82.68 s	120.24 s	22	***1.55 s***
	B-GVD	***16.08 s***	***2.52 s***	6	2.31 s
	Giraph	95.88 s	248.29 s	22	6.37 s
	GraphLab	188.59 s	77.0 s	22	7.57 s
USA Road	V-Centric	5.98 s	510.98 s	6262	0.57 s
	B-GVD	1.47 s	13.95 s	164	***1.00 s***
	B-2D	***1.41 s***	***1.94 s***	26	1.07 s
	Giraph	14.07 s	9518.99 s	6262	2.14 s
	Giraph++	16.81 s	24.00 s	12	2.54 s
	GraphLab	18.27 s	2982.3 s	6262	3.41 s

(a) Performance of Hash-Min

		Load	Compute	Step #	Dump
USA Road	V-Centric	7.21 s	2832.26 s	10789	***1.81 s***
	B-GVD	1.87 s	118.75 s	751	2.46 s
	B-2D	***1.65 s***	***11.29 s***	59	2.58 s
	Giraph	16.68 s	11116.90 s	10789	4.54 s
	Giraph++	18.29 s	80.53 s	39	2.88 s
	GraphLab	19.38 s	9293 s	10789	4.02 s
Euro Road	V-Centric	5.47 s	708.56 s	6210	1.01 s
	B-GVD	1.61 s	68.56 s	440	1.74 s
	B-2D	***1.26 s***	***8.86 s***	55	1.85 s
	Giraph	12.51 s	12712.06 s	6210	***0.05 s***
	Giraph++	18.57 s	87.73 s	30	3.59 s
	GraphLab	16.48 s	3231.3 s	6210	3.17 s

(b) Performance of SSSP

		Load	Compute	Step #	Dump
WebUK	V-Centric	***71.89 s***	144.29 s	664	***0.89 s***
	B-GVD	100.72 s	***41.58 s***	71	2.64 s
	Giraph	530.78 s	1078.06 s	664	13.75 s
	GraphLab	435.95 s	424.2 s	664	15.12 s
Live-Journal	V-Centric	8.93 s	6.87 s	19	***0.37 s***
	B-GVD	***5.54 s***	***4.84 s***	6	0.65 s
	Giraph	37.77 s	17.89 s	19	1.6 s
	Giraph++	17.50 s	29.12 s	4	16.13 s
	GraphLab	16.46 s	8.9 s	19	2.15 s
USA Road	V-Centric	5.18 s	389.37 s	6263	***0.45 s***
	B-GVD	2.31 s	33.31 s	246	0.73 s
	B-2D	***1.51 s***	***4.02 s***	26	1.40 s
	Giraph	16.27 s	5866.19 s	6263	2.64 s
	Giraph++	18.43 s	34.43 s	18	2.51 s
	GraphLab	18.86 s	1558.6 s	6263	3.56 s

(c) Performance of Reachability

Figure 3.15: Performance of CC, SSSP and Reachability Computation

eral graphs. We compare with Giraph, GraphLab, and the graph-centric system Giraph++. For GraphLab, we use its synchronous mode since this chapter focuses on synchronous computing model. For Giraph++, we do not report the results for *WebUK* and *Friendster* since Giraph++ failed to partition these large graphs.

Figure 3.15(a) shows the results of CC computation on three representative graphs: *BTC* (skewed degree distribution), *Friendster* (relatively high average degree), and *USA* (large diameter). We obtain the following observations. First, *V-centric* is generally faster than Giraph and GraphLab, which shows that Blogel is more efficient than existing systems even for vertex-centric computing. Second, *B-GVD* (or *B-2D*) is tens of times faster than *V-centric*, which shows the superiority of our block-centric computing. Finally, *B-GVD* (or *B-2D*) is 1–2 orders of magnitude faster than Giraph++; this is because Blogel's block-centric algorithm works in B-mode where blocks communicates with each other directly, while Giraph++'s graph-centric paradigm does not support B-mode and communication still occurs between vertices.

Figure 3.15(b) shows the results of SSSP computation on two weighted road network graphs. We see that both *B-GVD* and *B-2D* are orders of magnitude faster than *V-Centric*, which can be explained by the huge difference in the number of supersteps taken by the different models. Giraph++ is also significantly faster than *V-Centric*, but it is still much slower than our *B-2D*. This result verifies that our block-centric model can effectively deal with graphs with large diameter. The result also shows that 2D partitioner allows more efficient block-centric parallel computing than GVD partitioner for spatial networks.

Figure 3.15(c) shows the results of reachability computation on the small-diameter graph, *LiveJoural*, and the large-diameter graphs, *WebUK* and *USA*. We set the source s to be a vertex that can reach most of the vertices in the input graph and set $t = -1$, which means that the actual computation is BFS from s. As Figure 3.15(c) shows, our block-centric system, *B-GVD* or *B-2D*, is one to two orders of magnitude faster than the

		Load	Compute *lpr*(*v*) & *br*(*b*)	Dump
WebUK	**B-GVD**	105.10 s	44.98 s	385.45 s
	B-URL	124.52 s	17.39 s	395.93 s
WebBase	**B-GVD**	23.93 s	16.68 s	53.79 s
	B-URL	20.27 s	40.32 s	45.36 s

(a) Performance of BlockRank Computation

		Load	Step #	Per-Step Time	Dump
WebUK	**V-Centric**	71.37 s	89	29.99 s	4.16 s
	B-LDG	104.59 s	89	24.65 s	***2.04 s***
	B-GVD	***47.21 s***	95	***18.63 s***	6.59 s
	B-URL	64.62 s	93	25.96 s	7.08 s
	Giraph	163.99 s	89	53.74 s	22.35 s
	GraphLab	245.62 s	89	48.43 s	16.34 s
WebBase	**V-Centric**	20.81 s	80	16.23 s	***2.77 s***
	B-LDG	28.30 s	80	9.64 s	3.62 s
	B-GVD	11.51 s	90	4.99 s	6.92 s
	B-URL	***6.39 s***	84	***2.86 s***	5.15 s
	Giraph	61.41 s	80	12.67 s	16.30 s
	GraphLab	79.91 s	80	20.04 s	14.92 s

(b) Performance of PageRank Computation

		Load	Step #	Per-step Time	Dump
WebUK	**V-Centric**	***111.45 s***	92	31.16 s	***1.99 s***
	B-GVD	113.63 s	92	16.89 s	4.94 s
	B-URL	120.74 s	92	***9.30 s***	4.87 s
WebBase	**V-Centric**	30.74 s	92	16.53 s	***2.54 s***
	B-GVD	***25.38 s***	92	4.00 s	4.61 s
	B-URL	25.83 s	92	***1.20 s***	4.93 s
	Giraph++	149.49 s	92	7.02 s	26.47 s

(c) Performance of Giraph++'s Version

	Worker	Block #	Vertex #
WebUK	0 – 150	1,693 – 1697	884,987 – 884,988
WebBase	0 – 150	4,930 – 4,931	782,398 – 782,399

(d) # of Blocks/Vertices Per Worker (URL Partitioning)

Figure 3.16: Performance of PageRank Computation

vertex-centric systems, *V-Centric*, Giraph and GraphLab, for processing the large-diameter graphs. But the improvement is limited for the small-diameter graph, since *LiveJoural* is not very large and the vertex-centric systems are already very fast on a small-diameter graph of medium size. Compared with Giraph++, *B-GVD* is significantly more efficient for processing *LiveJoural* and *B-2D* is much faster for processing *USA*, while Giraph++ failed to run on *WebUK*.

For PageRank, we use the two web graphs *WebUK* and *WebBase*. We use both the URL-based partitioner and the GVD partitioner for graph partitioning. Figure 3.16(d) shows the number of blocks/vertices per worker using URL partitioning, where we see that URL partitioning achieves more balanced workload and less number of blocks than GVD partitioning (cf. Figure 3.9). This shows that background knowledge about the graph data can usually offer a higher quality partitioning than a general method.

We report the time of computing local PageRank and BlockRank by Blogel's B-mode in Figure 3.16(a). We present the average time of a superstep and the total number of supersteps of computing PageRank using different systems in Figure 3.16(b). We also run Blogel's block-centric V-mode with the input graph partitioned by LDG partitioning [45], denoted by *B-LDG* in Figure 3.16(b). We remark that LDG cannot be used in Blogel's VB-mode and B-mode, since a partition obtained by LDG is not guaranteed to be connected.

The results show that though running V-mode, block-centric computing is still significantly faster than vertex-centric computing (i.e., *V-Centric*, Giraph and GraphLab). Note that *B-GVD* and *B-URL* are also running block-centric V-mode in Figure 3.16(b). Thus, the result also reveals that our GVD partitioner leads to more efficient distributed computing than LDG. *B-URL* is comparable with *B-LDG* on *WebUK*, but is significantly faster than *B-LDG* on *WebBase*. The superior performance of *B-GVD* and *B-URL* is mainly because the GVD and URL partitioners achieve greater reduction in the number of cross-worker edges than LDG, which results in less number of messages exchanged through the network.

We also notice that the number of supersteps of the block-centric algorithm is more than that of the vertex-centric algorithm, which is mainly due to the fact that the PageRank initialization formula of [20], i.e., $pr(v) = lpr(v) \cdot br(block(v))$, is not effective. We may improve the algorithm by specifying another initialization formula, but this is not the focus of this chapter.

Another kind of PageRank algorithm is adopted in Giraph++'s paper [49], which is based on the accumulative iterative update approach of [58]. To make a fair comparison with Giraph++, we also developed a Blogel vertex-centric counterpart, and a block-centric counterpart that runs in VB-mode. As Figure 3.16(c) shows, Blogel's block-centric computing (i.e., *B-GVD* or *B-URL*) is significantly faster than its vertex-centric counterpart (i.e., *V-Centric*) when accumulative iterative update is applied. We can only compare with Giraph++ on *WebBase* since Giraph++ failed to partition *WebUK*. Figure 3.16(c) shows that while Giraph++ is twice faster than *V-Centric*, it is still much slower than *B-GVD* and *B-URL*.

3.8 Conclusions

We presented a block-centric framework, called Blogel, and showed that Blogel is significantly faster than existing distributed graph computing systems [1, 30, 17, 49], for processing large graphs with adverse graph characteristics such as skewed degree distribution, high average degree, and large diameter. We also showed that Blogel's partitioners generate high-quality blocks and are much faster than the state-of-the-art graph partitioning methods [45, 49].

Chapter 4

Pregel Algorithms with Performance Guarantee

Graphs in real life applications are often huge, such as the Web graph and various social networks. These massive graphs are often stored and processed in distributed sites. In this chapter, we study graph algorithms that adopt Google's Pregel, an iterative vertex-centric framework for graph processing in the Cloud. We first identify a set of desirable properties of an efficient Pregel algorithm, such as linear space, communication and computation cost per iteration, and logarithmic number of iterations. We define such an algorithm as a *practical Pregel algorithm* (PPA). We then propose PPAs for computing *connected components* (CCs), *biconnected components* (BCCs) and *strongly connected components* (SCCs). The PPAs for computing BCCs and SCCs use the PPAs of many fundamental graph problems as building blocks, which are of interest by themselves. Extensive experiments over large real graphs verified the efficiency of our algorithms.

The rest of this chapter is organized as follows. Section 4.1 reviews Pregel and related work. We define PPA in Section 4.2. Sections 4.3-4.5 discuss algorithms for CCs, BCCs and SCCs. Then, we report the experimental results in Section 4.6 and conclude in Section 4.7.

4.1 Related Work

Pregel [33]. Pregel is designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to different machines in a cluster, where each vertex v is associated with its adjacency list (i.e., the set of v's neighbors). A program in Pregel implements a user-defined *compute*() function and proceeds in iterations (called *supersteps*). In each superstep, the program calls *compute*() for each active vertex. The *compute*() function performs the user-specified task for a vertex v, such as processing v's incoming messages (sent in the previous superstep), sending messages to other vertices (to be received in the next superstep), and making v vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Pregel numbers the supersteps so that a user may use the current superstep number when implementing the algorithm logic in the *compute*() function. As a result, a Pregel algorithm can perform different operations in different supersteps by branching on the current superstep number.

Pregel allows users to implement a *combine*() function, which specifies how to combine messages that are sent from a machine M_i to the same vertex v in a machine M_j. These messages are combined into a single message, which is then sent from M_i to v in M_j. Combiner is applied only when commutative and associative operations are to be applied to the messages. For example, in Pregel's PageRank algorithm [33], messages from machine M_i that are to be sent to the same target vertex in machine M_j can be combined into a single message that equals their sum, since the target vertex is only interested in the sum of the messages. Pregel also supports aggregator, which is useful 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.

Pregel Algorithms. Besides this paper and Google's original paper on Pregel [33], we are only aware of two other papers studying Pregel algorithms, [36] and [41]. However, the algorithms are designed on a best-effort basis without any formal analysis on their complexity. Specifically, [36] aims at demonstrating that the Pregel model can be adopted to solve many graph problems in social network analysis, while [41] focuses on optimization techniques that overcome some performance bottlenecks caused by straightforward Pregel implementations.

MapReduce [11]. MapReduce, and its open-source implementation Hadoop, is another distributed system popularly used for large scale graph processing [22, 23, 28, 37, 46]. However, many graph algorithms are intrinsically iterative, in which case a Pregel program is much more efficient than its MapReduce counterpart. This is because each MapReduce job can only perform one iteration of graph computation, and it requires reading the entire graph from a distributed file system (DFS) and writing the processed graph back, which leads to excessive IO cost. Furthermore, a MapReduce job also needs to exchange the adjacency lists of vertices through the network, which results in heavy communication. In contrast, Pregel keeps vertices (along with their adjacency lists) in each local machine that processes their computation, and uses only message passing to exchange vertex states.

The "think like a vertex" programming paradigm also makes it easier for programmers to design graph algorithms in Pregel than in MapReduce.

GraphLab [30] and PowerGraph [17]. GraphLab [30] is another vertex-centric distributed graph computing system but it follows a different design from Pregel. GraphLab supports both synchronous and asynchronous executions. However, asynchronous execution does not have the concept of superstep number and hence cannot support algorithms that branch to different operations at different supersteps, such as the S-V algorithm in Section 4.3.2. Moreover, since GraphLab is mainly designed for asynchronous execution, its synchronous mode is not as expressive as Pregel. For example, since GraphLab only allows a vertex to access the states of its adjacent vertices and edges, it cannot support algorithms where a vertex needs to communicate with a non-neighbor. Another limitation of GraphLab is that it does not support graph mutations.

GraphLab 2.2, i.e., PowerGraph [17], partitions a graph by edges rather than by vertices in order to address imbalanced workload caused by high-degree vertices. However, a more complicated edge-centric Gather-Apply-Scatter (GAS) computing model should be used, which compromises user-friendliness.

PRAM. The PRAM model assumes that there are many processors and a shared memory. PRAM algorithms have been proposed for computing CCs [44], BCCs [48], and SCCs [4, 5, 15]. However, the PRAM model is not suitable for Cloud environments that are built on shared-nothing architectures. Furthermore, unlike Pregel and MapReduce, PRAM algorithms are not fault tolerant.

However, the ideas of many PRAM algorithms can be applied to design efficient Pregel algorithms. In Section 4.3.2, we shall demonstrate how the PRAM algorithm of [44] for computing CCs can be translated into a PPA. In Section 4.4.1, we describe the idea of [48]'s PRAM algorithm for computing BCCs, which will be used to design a PPA for computing BCCs.

4.2 Practical Pregel Algorithms

We now define some frequently used notations and introduce the notion of practical Pregel algorithms.

Notations. Given a graph $G = (V, E)$, we denote the number of vertices $|V|$ by n, and the number of edges $|E|$ by m. We also denote the *diameter* of G by δ. For an undirected graph, we denote the set of *neighbors* of a vertex v by $\Gamma(v)$ and the *degree* of v by $d(v) = |\Gamma(v)|$. For a directed graph, we denote the set of *in-neighbors* and *out-neighbors* of v by $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, and the *in-degree* and *out-degree* of v by $d_{in}(v) = |\Gamma_{in}(v)|$ and

$d_{out}(v) = |\Gamma_{out}(v)|$, respectively.

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 $O(d_{in}(v) + d_{out}(v))$) space of storage.

2. *Linear computation cost:* the time complexity of the $compute()$ function for each vertex v is $O(d(v))$ (or $O(d_{in}(v) + d_{out}(v))$).

3. *Linear communication cost:* at each superstep, the size 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 $O(\log n)$ supersteps.

Constraints 1-3 offers good load balancing and linear cost at each superstep, while Constraint 4 controls the total running time. As we shall see in later sections, some algorithms satisfying Constraints 1-3 require $O(\delta)$ rounds. Since many large real graphs have a small diameter δ, especially for social networks due to the small world phenomenon, we consider algorithms requiring $O(\delta)$ rounds also satisfying Constraint 4 if $\delta \leq \log n$ for the input graph.

For some problems, the per-vertex requirements of BPPA can be too strict, and we can only achieve *overall linear space usage, computation and communication cost* (still in $O(\log n)$ rounds). We call a Pregel algorithm that satisfies these constraints simply as a **practical Pregel algorithm** (**PPA**).

Motivation. We define BPPA and PPA in order to characterize a set of Pregel algorithms that can run efficiently in practice. Apart from the algorithms proposed in this chapter, other Pregel algorithms, e.g., the four demo algorithms in the Pregel paper [33], also have these characteristics (we can show that they are BPPAs): PageRank (constant supersteps), single-source shortest paths ($O(\delta)$ supersteps), bipartite matching ($O(\log n)$ supersteps), and semi-clustering (constant supersteps). However, these existing Pregel algorithms are designed on a best-effort basis and there is no formal performance requirement to be met or design rule to be followed. For example, while the Pregel algorithm developed in [36] for diameter estimation is an $O(\delta)$-superstep BPPA, the Pregel algorithm for triangle counting and clustering coefficient computation in [36] is not a PPA. Specifically, in superstep 1 of the triangle counting algorithm, a vertex sends a message for each pair of neighbors, leading to a quadratic number of messages to be buffered and sent. With the concept of PPA/BPPA in mind, users of the triangle counting algorithm can then be aware of the scalability limitation when applying this algorithm. The requirements of PPAs/BPPAs also serve as a guideline for programmers/researchers who want to develop efficient Pregel algorithms, and they may use existing PPAs as building blocks in their algorithms.

In the next three sections, we present PPAs/BPPAs for three fundamental graph connectivity problems. We also demonstrate how the PPAs/BPPAs of some fundamental graph problems can be used as building blocks to develop a more sophisticated PPA/BPPA for solving other graph problems such as computing BCCs.

4.3 Connected Components

In this section, we present two Pregel algorithms for computing CCs. Section 4.3.1 presents a BPPA that requires $O(\delta)$ supersteps. This algorithm works well on many real world graphs with a small diameter. However, it can be very slow on large-diameter graphs, such as spatial networks for which $\delta \approx O(\sqrt{n})$. We present an $O(\log n)$-superstep PPA in Section 4.3.2 to handle such graphs.

4.3.1 The Hash-Min PPA

Before presenting the $O(\delta)$-superstep BPPA for computing CCs, we first define some graph notations. We assume that all vertices in a graph G are assigned a unique ID. For convenience of discussion, we simply use v to refer to the ID of vertex v, and thus, the expression $u < v$ means that u's vertex ID is smaller than v's. We define the *color* of a (strongly) connected component in G to be the smallest vertex among all vertices in the component. The color of a vertex v, denoted by $color(v)$, is defined as the color of the component that contains v, and so, all vertices in G with the same color constitute a component.

A MapReduce algorithm, called Hash-Min, was proposed for computing CCs recently [37]. The idea of the algorithm is to broadcast the smallest vertex (ID) seen so far by each vertex v, denoted by $min(v)$; when the process converges, $min(v) = color(v)$ for all v. We now propose a BPPA counterpart as follows.

In Superstep 1, each vertex v initializes $min(v)$ as the smallest vertex in the set $(\{v\} \cup \Gamma(v))$, sends $min(v)$ to all v's neighbors and votes to halt. In each subsequent superstep, a vertex v obtains the smallest vertex from the incoming messages, denoted by u. If $u < v$, v sets $min(v) = u$ and sends $min(v)$ it to all its neighbors. Finally, v votes to halt.

We prove that the algorithm is a BPPA as follows. For any CC, it takes at most δ supersteps for the ID of the smallest vertex to reach all the vertices in the CC, and in each

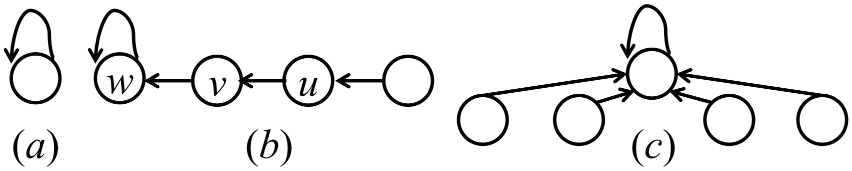


Figure 4.1: Forest structure of Shiloach-Vishkin's algorithm

superstep, each vertex v takes at most $O(d(v))$ time to compute $min(v)$ and sends/receives $O(d(v))$ messages each using $O(1)$ space.

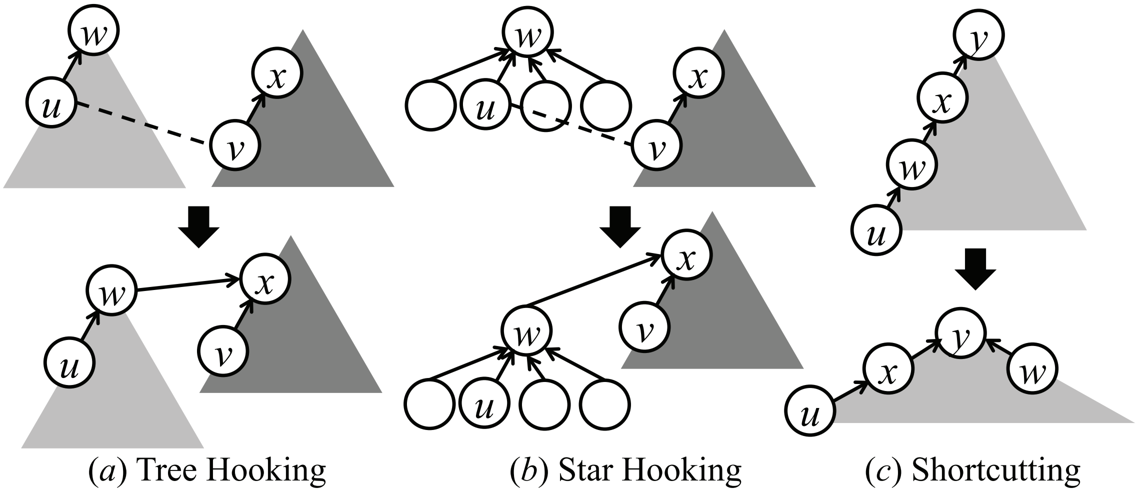
Three other MapReduce algorithms were also proposed in [37] for computing CCs. However, they require that each vertex maintain a set whose size can be as large as the size of its CC, and that the whole set be sent to some vertices. Thus, they cannot be translated into efficient Pregel implementations due to the highly skewed communication and computation, and the excessive space cost.

4.3.2 The S-V PPA

Our Contributions. We propose an $O(\log n)$-superstep PPA based on the S-V algorithm [44]. We note that the state-of-the-art distributed algorithms can only achieve $O(\log n)$ iterations in expectation on some types of graphs [37], while [37] also claims that the requirement of concurrent writes makes the S-V algorithm difficult to be translated to MapReduce (similarly to Pregel). We show that a direct translation of the S-V algorithm to Pregel is incorrect, and then change the algorithmic logic to obtain an $O(\log n)$-superstep PPA for CC computation.

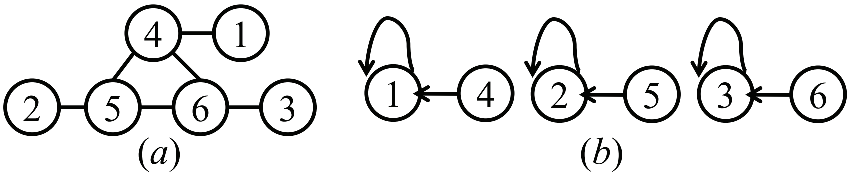
Algorithm Overview. In the S-V algorithm, each vertex u maintains a pointer $D[u]$. Initially, $D[u] = u$, forming a self loop as shown Figure 4.1(a). Throughout the algorithm, vertices are organized by a forest such that all vertices in each tree in the forest belong to the same CC. The tree definition is relaxed a bit here to allow the tree root w to have a self-loop (see Figures 4.1(b) and 4.1(c)), i.e., $D[w] = w$; while $D[v]$ of any other vertex v in the tree points to v's parent.

The S-V algorithm proceeds in rounds, and in each round, the pointers are updated in three steps (illustrated in Figure 4.2): (1)*tree hooking*: for each edge (u, v), if u's parent $w = D[u]$ is a tree root, hook w as a child of v's parent $D[v]$ (i.e., merge the tree rooted at w into v's tree); (2)*star hooking*: for each edge (u, v), if u is in a star (see Figure 4.1(c) for an example of star), hook the star to v's tree as Step (1) does; (3)*shortcutting*: for each vertex v, move vertex v and its descendants closer to the tree root, by hooking v to the parent of v's parent, i.e., setting $D[v] = D[D[v]]$. The algorithm ends when every vertex



(a) Tree Hooking (b) Star Hooking (c) Shortcutting

Figure 4.2: Tree hooking, star hooking, and shortcutting



(a) (b)

Figure 4.3: Illustration of the change to star hooking

is in a star.

We perform tree hooking in Step (1) only if $D[v] < D[u]$, so that if u's tree is hooked to v's tree due to edge (u, v), then edge (v, u) will not hook v's tree to u's tree again.

The condition "$D[v] < D[u]$" is not required for star hooking, since if u's tree is hooked to v's tree, v's tree cannot be a star (e.g., in Figure 4.2(b), after hooking, u is two hops away from x). However, in Pregel's computing model, Step (2) cannot be processed. Consider the graph shown in Figure 4.3(a), and suppose that we obtain the three stars in Figure 4.3(b) right after Step (1). If the one-directional condition is not required, setting $D[D[u]]$ as $D[v]$ makes $D[1] = 2$, $D[2] = 3$ and $D[3] = 1$ through the edges $(4, 5)$, $(5, 6)$ and $(6, 4)$, thus forming a cycle and violating the tree formation required by the algorithm. Such a problem does not exist in the PRAM model since the values of $D[u]$ and $D[v]$ are immediately updated after each write operation, while in Pregel the values are those received from the previous superstep.

To address this problem, we examine the algorithm logic in Pregel execution and find that if we always require $v_a < v_b$ when setting $D[v_a] \leftarrow v_b$ during hooking, we can prove that the pointer values monotonically decrease, and thus $D[v] = color(v)$ for any vertex v when the algorithm terminates. Based on this result, we change the algorithm by requiring

"$D[v] < D[u]$" for star hooking. Then, the S-V algorithm can be translated into a Pregel algorithm by exchanging messages to update the pointers $D[.]$ following the three steps in Figure 4.2, until every vertex v is in a star.

Algorithm 1 presents the original S-V algorithm for the PRAM model, where "par($o \in S$) $\{op(o)\}$" means that the operation $op(o)$ runs in parallel for all objects o in set S. Pointers $D[.]$ are updated by examining edge links in parallel. Initially, for each vertex u, $D[u]$ is set as a neighbor $v < u$ if such a v exists (Line 2). Then, in each round, the pointers are updated by tree hooking (Line 5), star hooking (Line 7) and shortcutting (Line 8).

Each vertex u is also associated with a flag $star[u]$ indicating whether it is in a star. The algorithm terminates when all trees become stars, and each star corresponds to a CC. The value of $star[u]$ is required in two places in the while-loop: Line 3 and Line 7. Algorithm 2 shows the PRAM algorithm for computing $star[u]$ for all $u \in V$ according to the current pointer setting, based on the fact that a vertex u with $D[u] \neq D[D[u]]$ invalidates a tree from being a star (see vertex u in Figure 4.1(b)).

We now illustrate how to translate the steps of Algorithm 1 into Pregel.

Tree Hooking in Pregel. We compute Lines 4–5 of Algorithm 1 in four supersteps: (1)each vertex u sends request to $w = D[u]$ for $D[w]$; (2)each vertex w responds to u by sending $D[w]$; and meanwhile, each vertex v sends $D[v]$ to all v's neighbors; (3)each vertex u obtains $D[w]$ and $D[v]$ from incoming messages and evaluates the if-condition, and then an arbitrary v that satisfies the condition (if it exists) is chosen and $D[v]$ is sent to w; (4)each vertex w that receives messages sets $D[w] \leftarrow D[v]$ using an arbitrary message $D[v]$.

Star Hooking in Pregel. We compute Lines 6–7 of Algorithm 1 similarly. However, the condition "$D[u] \neq D[v]$" in Line 7 should be changed to "$D[v] < D[u]$".

Computing $star[u]$ in Pregel. We compute Algorithm 2 in five supersteps: (1)each vertex u sets $star[u] \leftarrow true$, and sends request to $w = D[u]$ for $D[w]$; (2)each vertex w responds by sending back $D[w]$; (3)each vertex v checks whether $D[u] = D[w]$; if not, it sets $star[u] \leftarrow false$ and notifies w and $D[w]$; (4)each vertex w that gets notified sets $star[w] \leftarrow false$. To process Line 5, in Superstep (3), we also make each vertex u send request to $w = D[u]$ for $star[w]$; in Superstep (4), we also make w respond by sending $star[w]$ (note that if w get notified, it must set $star[w]$ to be false first). Finally, in Superstep (5), each vertex u gets $star[w]$ and uses it to update $star[u]$.

Analysis. The other steps can be translated to Pregel similarly. To check the condition in Line 3 of Algorithm 1, we use an aggregator that computes the AND of $star[u]$ for all vertices u. The algorithm terminates if its value equals $true$.

The correctness of this PPA can be justified as follows. Since we always require $v_a <$

Algorithm 1: The Shiloach-Vishkin Algorithm

```
par($u \in V$) {$D[u] \leftarrow u$}
par($(u, v) \in E$) {if($v < u$) $D[u] \leftarrow v$}
while($\exists u : star[u] = false$) {
    par($(u, v) \in E$)
        if($D[D[u]]$=$D[u]$ and $D[v]$<$D[u]$){ $D[D[u]] \leftarrow D[v]$}
    par($(u, v) \in E$)
        if($star[u]$ && $D[u] \neq D[v]$){ $D[D[u]] \leftarrow D[v]$}
    par($u \in V$) $D[u] \leftarrow D[D[u]]$
}
```

Algorithm 2: Computing $star[u]$

```
par($u \in V$) {
    $star[u] \leftarrow true$
    if($D[u] \neq D[D[u]]$){     $star[u] \leftarrow false$;
        $star[D[u]] \leftarrow false$;  $star[D[D[u]]] \leftarrow false$}
    $star[u] \leftarrow star[D[u]]$
}
```

v_b when setting $D[v_a] \leftarrow v_b$ during hooking, the pointer values monotonically decrease, and thus $D[v] = color(v)$ for any vertex v when the algorithm terminates.

Since each step requires a constant number of supersteps, each round of the S-V algorithm uses a constant number of supersteps (14 in our implementation). Following an analysis similar to [44], the algorithm computes CCs in $O(\log n)$ rounds, and therefore we obtain a PPA for computing CCs. However, the algorithm is not a BPPA since a vertex w may become the parent of more than $d(w)$ vertices and hence receives/sends more than $d(w)$ messages at a superstep, though the overall number of messages is always bounded by $O(n)$ at each superstep.

The S-V based Pregel algorithm is an $O(\log n)$-superstep PPA, which can be proved as the original S-V algorithm computes CCs in $O(\log n)$ rounds [44], where each round is implemented in a constant number of supersteps. However, the algorithm is not a BPPA since a vertex v may become the parent of more than $d(v)$ vertices and hence receives/sends more than $d(v)$ messages in a superstep, though the overall number of messages in each superstep is always bounded by $O(n)$.

Extension for Computing Spanning Tree. The S-V algorithm can be modified to obtain an $O(\log n)$-superstep PPA that computes a spanning forest of a graph. The main idea is to

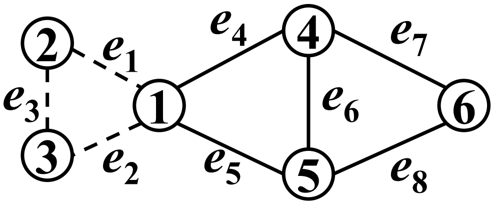


Figure 4.4: BCC illustration

mark an edge (u, v) as a tree-edge if a hooking operation is performed due to (u, v), which is straightforward for Line 2 of Algorithm 1. For Lines 4 and 6 that perform $D[D[u]] \leftarrow D[v]$, the last superstep is for $w = D[u]$ to pick an arbitrary message $D[v]$ sent by some v, and set $D[w] \leftarrow D[v]$. In this case, w needs to notify both u and v to mark their edge (u, v) as a tree-edge.

4.4 Bi-Connected Components

Our PPA for computing BCCs is based on the idea of the PRAM algorithm in [48], but we make the following new contributions.

Our Contributions. To our knowledge, the problem of computing BCCs in Pregel has never been studied before. Thus, it is important to show that a Pregel algorithm with strong performance guarantee exists for this problem, which we will establish by proposing a PPA for computing BCCs. Second, existing studies on designing Pregel algorithms [37, 36] often neglect the rich body of PRAM algorithms. Our PPA for computing BCCs demonstrates that some ideas from the PRAM algorithms can be applied to design Pregel algorithms. Third, though the main idea is based on [48], the design of our PPA for BCC computation is non-trivial. Specifically, our BCC algorithm is composed of a number of building blocks, and to ensure that our final algorithm is a PPA, we devise a PPA for each building block. Finally, these building blocks used in our BCC algorithm are themselves fundamental graph operations that are useful to the design of many other distributed graph algorithms. Therefore, we study them in greater depth and carefully design a PPA for each of them, which are often much simpler than the existing PRAM algorithms.

4.4.1 BCC and Its PRAM Algorithm

Bi-connected Component (BCC). A BCC of an undirected graph G is a maximal subgraph of G that remains connected after removing one arbitrary vertex. We illustrate the concept of BCC using the graph shown in Figure 4.4, where the dashed edges constitute one BCC, and the solid edges constitute another. Let R be the equivalence relation on the set of edges of G such that $e_1 R e_2$ iff $e_1 = e_2$ or e_1 and e_2 appear together in some simple cycle, then R defines the BCCs of G. For example, in Figure 4.4, edges $(4, 5)$ and $(5, 6)$ are in cycle $(4, 5, 6, 4)$, but there is no simple cycle containing both $(4, 5)$ and $(1, 2)$. A vertex is called an *articulation point* if it belongs to more than one BCC, such as vertex 1 in Figure 4.4. The removal of an articulation point disconnects the connected components containing it.

Tarjan-Vishkin's PRAM Algorithm. If we construct a new graph G' whose vertices correspond to the edges of G, and an edge (e_1, e_2) exists in G' iff $e_1 R e_2$, then the CCs of G' correspond to the BCCs of G. However, the number of edges in G' can be superlinear to m. Tarjan-Vishkin's (T-V) PRAM algorithm [48] constructs a concise graph G^* containing a small subset of the edges in G', whose size is bounded by $O(m)$. They prove that it is sufficient to compute the CCs of G^* to obtain the BCCs of G.

Since our PPA for computing BCCs is also based on this idea, we first present the definition of G^* below. Assume that $G = (V, E)$ is connected and T is a spanning tree of G. Also, assume that T is rooted and each vertex u in T is assigned a pre-order number $pre(u)$. Let $p(u)$ be the parent of a vertex u in T. We construct $G^* = (V^*, E^*)$ as follows. First, we set $V^* = E$. Then, we add an edge (e_1, e_2) to E^*, where $e_1, e_2 \in E$, iff e_1 and e_2 satisfy one of the following three cases:

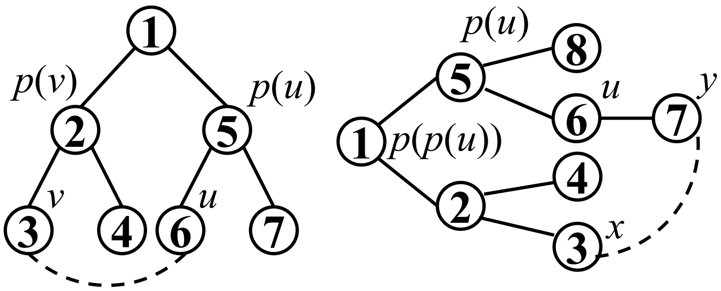
- Case 1: $e_1 = (p(u), u)$ is a tree edge in T, and $e_2 = (v, u)$ is a non-tree edge (i.e., e_2 is not in T) with $pre(v) < pre(u)$.

- Case 2: $e_1 = (p(u), u)$ and $e_2 = (p(v), v)$ are two tree edges in T, u and v have no ascendant-descendant relationship in T, and $(u, v) \in E$.

- Case 3: $e_1 = (p(u), u)$ and $e_2 = (p(p(u)), p(u))$ are two tree edges in T, and $\exists (x, y) \in E$ s.t. x is a non-descendant of $p(u)$ in T and y is a descendant of u in T.

Figure 4.5 illustrates the three cases, where solid edges are tree edges in T, and dashed edges are non-tree edges in $(G - T)$. The vertices are labeled by their pre-order numbers. Case 1 is shown by $e_1 = (5, 6)$ and $e_2 = (3, 6)$ in Figure 4.5(a). Note that $e_1 R e_2$ since e_1 and e_2 are in a simple cycle $\langle 1, 2, 3, 6, 5, 1 \rangle$. Case 2 is also shown in Figure 4.5(a) by $e_1 = (5, 6)$ and $e_2 = (2, 3)$, and also e_1 and e_2 are in the same simple cycle. Case 3 is shown in Figure 4.5(b) by $e_1 = (5, 6)$ and $e_2 = (1, 5)$, where e_1 and e_2 are in the simple cycle $\langle 1, 2, 3, 7, 6, 5, 1 \rangle$.

Each non-tree edge (u, v) of G introduces at most one edge into G^* due to Case 1 (and Case 2), and each tree-edge $(p(u), u)$ introduces at most one edge due to Case 3. Therefore, $|E^*| = O(m)$.



(a) Case 1 & 2 (b) Case 3

Figure 4.5: Illustration of construction of G^*

4.4.2 PPA for Computing BCCs

Our PPA for computing BCCs is also based on the idea of Tarjan-Vishkin's algorithm, i.e., to construct the concise graph G^* and then compute the CCs of G^* to obtain the BCCs of G. Without loss of generality, we assume G is connected, as BCC computation in different CCs is independent and can be parallelized. To construct G^*, we first propose a set of building blocks in Sections 4.4.2- 4.4.2, and then in Section 4.4.2 we put everything together to obtain our final PPA for computing BCCs.

Spanning Tree Computation

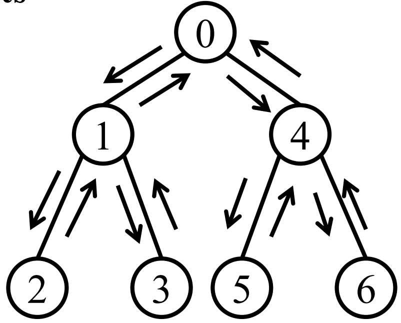
To construct G^*, we first need a spanning tree of G, denoted by T. We present an $O(\delta)$-superstep BPPA for spanning tree computation as follows.

The algorithm performs *breadth-first search* (BFS) and computes a spanning tree over an unweighted graph G from a source vertex s. Each vertex v in G maintains two fields, the parent of v, denoted by $p(v)$; and the shortest-path distance (or BFS level) of v from s, denoted by $dist(v)$. Initially, only s is active with $p(s) = null$ and $dist(s) = 0$, and $dist(v) = \infty$ for all other v. In Superstep 1, s sends $\langle s, dist(s)\rangle$ to all its neighbors, and votes to halt. In each subsequent superstep, if a vertex v receives any message, it first checks whether v has been visited before (i.e., whether $dist(v) < \infty$): if not, it updates $dist(v) = dist(u) + 1$ and $p(v) = u$ with an arbitrary message $\langle u, dist(u)\rangle$ received, and sends $\langle v, dist(v)\rangle$ to all v's neighbors. Finally, v votes to halt.

When the algorithm terminates, we obtain a tree edge $(p(v), v)$ from each vertex $v \neq s$, which constitute a spanning tree rooted at s. It is easy to see that the algorithm is an $O(\delta)$-superstep BPPA. In the case if G is disconnected, we first compute $color(v)$ for each vertex v using the algorithm described in Section 4.3.1, and then pick the vertex s with $s = color(s)$ as the source for each CC. Since, multi-source BFS is done in parallel, the overall number of supersteps is still $O(\delta)$. For processing graphs with a large diameter

Adjacency Lists

0: 1, 4
1: 0, 2, 3
2: 1
3: 1
4: 0, 5, 6
5: 4
6: 4



Eular Tour

(1) **(0, 1)** → (2) **(1, 2)** →
(3) **(2, 1)** → (4) **(1, 3)** →
(5) **(3, 1)** → (6) **(1, 0)** →
(7) **(0, 4)** → (8) **(4, 5)** →
(9) **(5, 4)** → (10) **(4, 6)** →
(11) **(6, 4)** → (12) **(4, 0)**

Figure 4.6: Euler tour

δ, as mentioned in Section 4.3.2, the S-V algorithm can be extended to give an $O(\log n)$-superstep PPA to compute the spanning tree T.

Pre-order Numbering

Consider the three cases for constructing the edges of G^* in Section 4.4.1. In Case 1, we need the pre-order number of each vertex v (i.e., $pre(v)$) in the spanning tree T. We present an $O(\log n)$-superstep BPPA to compute the pre-order numbers for all vertices in T. We also propose a symmetric BPPA for computing post-order numbers.

To compute the pre-order numbering, we first compute a Euler tour of the spanning tree T. A Euler tour is a representation of a tree which is useful in many parallel graph algorithms. The tree is viewed as a directed graph, where each tree edge (u, v) is considered as two directed edges (u, v) and (v, u), and a Euler tour of the tree is simply a Eulerian circuit of the directed graph, i.e., a trail that visits every edge exactly once, and ends at the same vertex where it starts.

We present an $O(\log n)$-superstep BPPA to compute the Euler tour of a tree T as follows.

BPPA for Computing Euler Tour. Assume that the neighbors of each vertex v are sorted according to their IDs, which is common for an adjacency list representation of a graph. For a vertex v, let $first(v)$ and $last(v)$ be the first and last neighbor of v in the sorted order; and for each neighbor u of v, if $u \neq last(v)$, let $next_v(u)$ be the neighbor of v next to u in the sorted adjacency list. We further define $next_v(last(v)) = first(v)$. Consider the example shown in Figure 4.6. For the adjacency list of vertex 4, we have $first(4) = 0$, $last(4) = 6$, $next_4(0) = 5$, $next_4(5) = 6$ and $next_4(6) = 0$.

If we translate $next_v(u) = w$ as specifying that the edge next to (u, v) is (v, w), we obtain a Euler tour of the tree. Referring to the example in Figure 4.6 again, where a Euler tour that starts and ends at vertex 0 is given. The next edge of $(2, 1)$ is $(1, 3)$, because $next_1(2) = 3$, while the next edge of $(6, 4)$ is $(4, 0)$ because $next_4(6) = 0$. In fact, starting from any vertex v and any neighbor u of v, $\langle(v, x = next_v(u))$, $(x, y = next_x(v))$, $(y, next_y(x))$, . . ., $(u, v)\rangle$ defines a Euler tour.

We present a 2-superstep BPPA to construct the Euler tour as follows: In Superstep 1, each vertex v sends message $\langle u, next_v(u)\rangle$ to each neighbor u; in Supertep 2, each vertex u receives the message $\langle u, next_v(u)\rangle$ sent from each neighbor v, and stores $next_v(u)$ with v in u's adjacency list. When the algorithm finishes, for each vertex u and each neighbor v, the next edge of (u, v) is obtained as $(v, next_v(u))$.

The algorithm requires a constant number of supersteps, and in each superstep, each vertex v sends/receives $O(d(v))$ messages (each using $O(1)$ space). By implementing $next_v(.)$ as a hash table associated with v, we can obtain $next_v(u)$ in $O(1)$ expected time given u.

After obtaining the Euler tour of T, which is a cycle of edges, we break it at some edge to obtain a list of edges. We then compute the pre-order and post-order numbers of the vertices in T from the list, using the list ranking operation. Since our BPPAs for pre-order and post-order numbering are based on list ranking, we first introduce the concept of list ranking and present an $O(\log n)$-superstep BPPA for list ranking below.

BPPA for List Ranking. Consider a linked list $\mathcal{L}$ with n objects, where each object v is associated with a value $val(v)$ and a link to its predecessor $pred(v)$. The object v at the head of $\mathcal{L}$ has $pred(v) = null$. For each object v in $\mathcal{L}$, let us define $sum(v)$ to be the sum of the values of all the objects from v following the predecessor link to the head. The *list ranking* problem computes $sum(v)$ for each object v. If $val(v) = 1$ for each v in $\mathcal{L}$, then $sum(v)$ is simply the rank of v in the list, i.e., the number of objects preceding v plus 1.

In list ranking, the objects in $\mathcal{L}$ are given in arbitrary order. We may regard $\mathcal{L}$ simply as a directed graph consisting of a single simple path. Albeit simple, list ranking is an important problem in parallel computing because it serves as a building block to many other parallel algorithms.

We now describe our BPPA for list ranking. Initially, each vertex v assigns $sum(v) = val(v)$. Then in each round, each vertex v does the following: If $pred(v) \neq null$, v sets $sum(v) = sum(v) + sum(pred(v))$ and $pred(v) = pred(pred(v))$; otherwise, v votes to halt. The if-branch is accomplished in three supersteps: (1)v sends a message to $u = pred(v)$ requesting for the values of $sum(u)$ and $pred(u)$; (2)u sends back the requested values to v; and (3)v updates $sum(v)$ and $pred(v)$. 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 4.7 illustrates how the algorithm works. Initially, objects v_1–v_5 form a linked

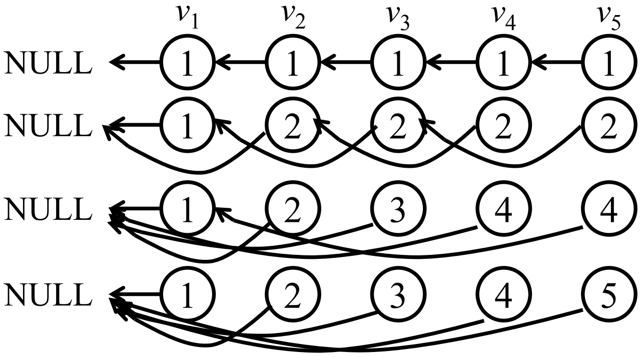


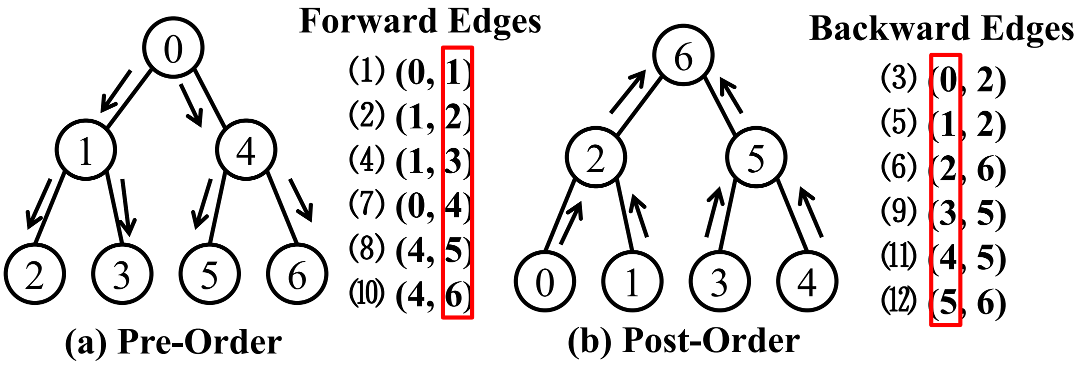
Figure 4.7: Illustration of BPPA for list ranking

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 $sum(v_5) = sum(v_5) + sum(v_4) = 1 + 1 = 2$ and $pred(v_5) = pred(v_4) = v_3$. One can verify the states of the other vertices similarly. In Round 2, we have $pred(v_5) = v_3$ and so we set $sum(v_5) = sum(v_5) + sum(v_3) = 2 + 2 = 4$ and $pred(v_5) = pred(v_3) = v_1$. In Round 3, we have $pred(v_5) = v_1$ and so we set $sum(v_5) = sum(v_5) + sum(v_1)$=4 + 1=5 and $pred(v_5) = pred(v_1) = null$. We can prove by induction that in Round i, we set $sum(v_j) = \sum_{k=j-2^i+1}^{j} val(v_k)$ and $pred(v_j) = v_{j-2^i}$. Furthermore, each object v_j sends at most one message to $v_{j-2^{i-1}}$ and receives at most one message from $v_{j+2^{i-1}}$. The algorithm is a BPPA because it terminates in $\log n$ rounds, and each object sends/receives at most one message per round.

Let us assume that we already obtain the Euler tour $\mathcal{P}$ of the spanning tree starting from s, given by $\mathcal{P} = \langle(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, s)\rangle$. We break the cycle into a list by setting $pred(s, v_1) = null$. We now consider how to compute the pre-order and post-order numbers from the list using list ranking.

Pre-Order and Post-Order Numbering. Depth-first traversal of a tree T generates pre-order or post-order numbers for the vertices in a tree, and the numbers are useful in many applications, such as deciding the ancestor-descendant relationship of two tree vertices, which we discuss in Section 4.4.2. Let $pre(v)$ and $post(v)$ be the pre-order and post-order number of each vertex v in T, respectively. We present a BPPA for computing pre-order and post-order numbers from the Euler tour $\mathcal{P}$ of the tree T below.

We formulate a list ranking problem by treating each edge $e \in \mathcal{P}$ as a vertex and setting $val(e) = 1$. After obtaining $sum(e)$ for each $e \in \mathcal{P}$, we mark the edges in $\mathcal{P}$ as forward/backward edges using a two-superstep BPPA: in Superstep 1, each vertex $e = (u, v)$ sends $sum(e)$ to $e' = (v, u)$; in Superstep 2, each vertex $e' = (v, u)$ receives $sum(e)$ from $e = (u, v)$, sets e' itself as a forward edge if $sum(e') < sum(e)$, and a backward



(a) Pre-Order
(b) Post-Order

Figure 4.8: Pre-order & post-order

edge otherwise. In Figure 4.6, edge $(1, 2)$ is a forward edge because its rank (i.e., 2) is smaller than that of $(2, 1)$ (i.e., 3), while edge (4, 0) is a backward edge since its rank (i.e., 12) is larger than that of $(0, 4)$ (i.e., 7).

To compute $pre(v)$, we run a second round of list ranking by setting $val(e) = 1$ for each forward edge e in $\mathcal{P}$ and $val(e') = 0$ for each backward edge e'. Then, for each forward edge $e = (u, v)$, we get $pre(v) = sum(e)$ for vertex v. We set $pre(s) = 0$ for tree root s. For example, Figure 4.8(a) shows the forward edges (u, v) in the order in $\mathcal{P}$, where vertices are already labeled with pre-order numbers. Obviously, the rank of (u, v) gives $pre(v)$.

To compute $post(v)$, we run list ranking by setting $val(e) = 0$ for each forward edge e and $val(e') = 1$ for each backward edge e' in $\mathcal{P}$. Then, for each backward edge $e' = (v, u)$, we get $post(v) = sum(e')$ for vertex v. We set $post(s) = n - 1$ for tree root s, where n is the number of vertices in the tree. If n is not known, we can easily compute n using an aggregator in Pregel with each vertex providing a value of 1. (In the more general case where G is a forest, the aggregator counts the number of vertices for each tree/component). For example, Figure 4.8(b) shows the backward edges (v, u) in the order in $\mathcal{P}$, and vertices are relabeled with post-order numbers. Obviously, the rank of (v, u) gives $post(v)$.

The algorithm correctly computes $pre(v)$/$post(v)$ for all vertices v, because each vertex v in the tree (except root s) has exactly one parent u defined by the forward/backward edge (u, v)/(v, u). Finally, the proof for BPPA follows directly from the fact that both Euler tour and list ranking can be computed by BPPAs.

Ancestor-Descendant Query

In Case 2 for constructing the edges of G^*, we need to decide whether two vertices u and v have ancestor-descendant relationship in the spanning tree T.

Let $pre(v)$ be the pre-order number of v and $nd(v)$ be the number of descendants of v in the tree. We show that if $pre(v)$ and $nd(v)$ is available for each vertex v in the tree, then an ancestor-descendant query can be answered in $O(1)$ time. Given u and v, following the definition of pre-order numbering, we have: u is an ancestor of v iff $pre(u) \leq pre(v) < pre(u) + nd(u)$. For vertex 1 in Figure 4.8(a), we have $pre(1) = 1$ and $nd(1) = 3$, and therefore any vertex v with $1 \leq pre(v){<}1{+}3$ (i.e., vertices 1, 2 and 3) is a descendants of vertex 1.

We have presented a BPPA to compute $pre(v)$ in Section 4.4.2. We now show that $nd(v)$ can be obtained in the same process: for each forward edge $e = (u, v)$, we set $nd(v) = sum(e'){-}sum(e){+}1$ where e' is the backward edge (v, u). For tree root s, we set $nd(s) = n$. For example, we compute $nd(1) = sum(1, 0){-}sum(0, 1){+}1 = 3{-}1{+}1 = 3$ for vertex 1 in Figure 4.8(a).

Case 3 Condition Checking

With the PPA/BPPA proposed in the previous subsections, Case 1 and Case 2 in Section 4.4.1 can now be checked in a constant number of supersteps by exchanging messages with neighbors. Case 3, however, is far more complicated to handle as vertices other than direct neighbors are involved.

We now develop a PPA for handling Case 3 as follows. We first need to compute two more fields for each vertex v, which are defined recursively as follows:

- $min(v)$: the minimum of (1)$pre(v)$, (2)$min(u)$ for all of v's children u, and (3)$pre(w)$ for all non-tree edges (v, w).
- $max(v)$: the maximum of (1)$pre(v)$, (2)$max(u)$ for all of v's children u, and (3)$pre(w)$ for all non-tree edges (v, w).

Let $desc(v)$ be the descendants of v (including v itself) and $\Gamma_{desc}(v)$ be the set of vertices connected to any vertex in $desc(v)$ by a non-tree edge. Intuitively, $min(v)$ (or $max(v)$) is the smallest (or largest) pre-order number among $desc(v) \cup \Gamma_{desc}(v)$.

In Case 3, (x, y) exists iff $x \in \Gamma_{desc}(u) - desc(v)$. Since x is not a descendant of $p(u)$, either $pre(x) < pre(p(u))$ which implies $min(u) < pre(p(u))$, or $pre(x) \geq pre(p(u)) + nd(p(u))$ which implies $max(u) \geq pre(p(u)) + nd(p(u))$. To summarize, Case 3 holds for u iff $min(u) < pre(p(u))$ or $max(u) \geq pre(p(u)) + nd(p(u))$.

When $pre(v)$, $nd(v)$, $min(v)$ and $max(v)$ are available for each vertex v, all the three cases can be handled using $O(1)$ supersteps. We now show how to compute $min(v)$ for each v by a PPA in $O(\log n)$ supersteps (computing $max(v)$ is symmetric).

For ease of presentation, let us simply use v to denote $pre(v)$. We further define $local(v)$ to be the minimum among v and all the neighbors connected to v by a non-tree edge. Note that $min(v)$ is just the minimum of $local(u)$ among all of v's descendants u.

We compute $min(v)$ in $O(\log n)$ rounds. At the beginning of the i-th round, each vertex $v = c \cdot 2^i$ (c is a natural number) maintains a field $global(c \cdot 2^i) = \min\{local(u) : c \cdot 2^i \leq u < (c+1)2^i\}$. Then in the $(i+1)$-th round, for each vertex $v = c \cdot 2^{i+1}$ we can simply update $global(v) = \min\{global(v), global(v + 2^i)\}$, i.e., merging the results from two consecutive segments of length 2^i. Here, each round can be done by a three-superstep PPA: (1) each v requests $global(v + 2^i)$ from $(v + 2^i)$; (2) $(v + 2^i)$ responds by sending $global(v + 2^i)$ to v; (3) v receives $global(v + 2^i)$ to update $global(v)$. Initially, $global(v) = local(v)$ for each v, and $local(v)$ can be computed similarly, by requesting $pre(u)$ from each neighbor u connected to v by a non-tree edge.

Given a vertex v, the descendants of v in T are $\{v, v+1, \ldots, v + nd(v) - 1\}$. At the beginning of the i-th round, we define $little(v)$ (and respectively, $big(v)$) to be the first (and respectively, the last) descendant that is a multiple of 2^i. Figure 4.9 illustrates the concept of $little(v)$ and $big(v)$.

We maintain the following invariant for each round:

$$\begin{aligned} min(v) \quad = \quad & \min\{local(u) : u \in [v, little(v)) \cup \\ & \quad [big(v), v + nd(v) - 1]\}. \end{aligned} \tag{4.1}$$

Obviously, the correct value of $min(v)$ is computed when $little(v) = big(v)$, which must happen for some value of i as i goes from 0 to $\lfloor \log_2 n \rfloor$. We perform the following operations for each v in the i-th round, which maintains the invariant given by Equation (4.1):

- If $little(v) < big(v)$ and $little(v)$ is not a multiple of 2^{i+1}, set $min(v) = \min\{min(v), global(little$ and then set $little(v) = little(v) + 2^i$.

- If $little(v) < big(v)$ and $big(v)$ is not a multiple of 2^{i+1}, set $min(v) = \min\{min(v), global(big(v) -$ $2^i)\}$, and then set $big(v) = big(v) - 2^i$.

We do not update $little(v)$ (or $big(v)$) if it is a multiple of 2^{i+1}, so that in the $(i+1)$-th round it is aligned with $c \cdot 2^{i+1}$. We update $little(v)$, $big(v)$, and $min(v)$ by Pregel operations similar to those for updating $global(v)$.

Integration of Building Blocks

Refer to Section 4.4.1 again, when computing the CCs of G^*, we only need to consider those vertices of G^* that correspond to the tree edges in T. This is because other vertices

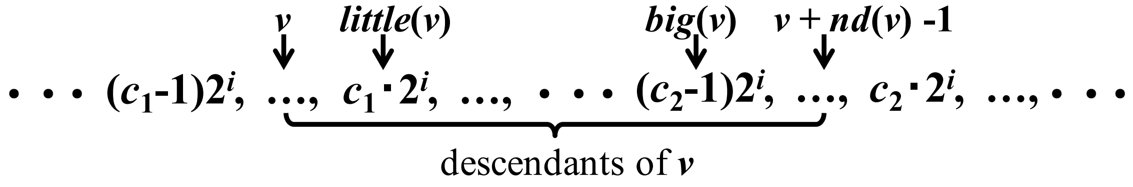

v
little(v)
big(v)
v + nd(v) -1
• • • (c1-1)2i, ..., c1·2i, ..., • • • (c2-1)2i, ..., c2·2i, ..., • • •
descendants of v

Figure 4.9: Illustration of computing $min(v)$

of G^* correspond to the non-tree edges e_2 of Case 1, and hence can be assigned to the CC of the corresponding e_1 later on.

We now integrate all the building blocks into a PPA for computing BCCs. The algorithm consists of a sequence of PPA tasks: (1)*HashMin*: to compute $color(v)$ for all $v \in V$ using the BPPA of Section 4.3.1; (2)*BFS* or *S-V*: to compute a spanning forest of G using the BPPA of Section 4.4.2 with sources $\{s \in V | color(s) = s\}$; alternatively, we may obtain the spanning forest using the S-V algorithm of Section 4.3.2, denoted by *S-V*; (3)*EulerTour*: to construct Euler tours from the spanning forest using the BPPA of Section 4.4.2; (4)*ListRank1*: to break each Euler tour into a list and mark each edge as forward/backward using list ranking (see Section 4.4.2); (5)*ListRank2*: using the edge forward/backward marks to compute $pre(v)$ and $nd(v)$ for each $v \in V$ using list ranking (see Section 4.4.2); (6)*MinMax*: to compute $min(v)$ and $max(v)$ for each $v \in V$ using the PPA of Section 4.4.2; (7)*AuxGraph*: to construct G^* using $pre(v)$, $nd(v)$, $min(v)$ and $max(v)$ information; this is a constant-superstep BPPA since all three cases for edge construction can be checked in a constant number of supersteps. (8)*HashMin2* or *S-V2*: to compute the CCs of G^* using *HashMin*, but only consider tree edges; alternatively, we may use the *S-V* algorithm for CC computation; (9)*Case1Mark*: to decide the BCCs of the non-tree edges in G^* using Case 1.

This algorithm is a PPA since each of its tasks is a BPPA/PPA.

4.5 Strongly Connected Components

In this section, we present two novel Pregel algorithms that compute *strongly connected components* (**SCC**s) from a directed graph $G = (V, E)$. Let $SCC(v)$ be the SCC that contains v, and let $Out(v)$ (and $In(v)$) be the set of vertices that can be reached from v (and respectively, that can reach v) in G. Some PRAM algorithms [15, 5, 4] were designed based on the observation that $SCC(v) = Out(v) \cap In(v)$ (see Figure 4.10(a)). They compute $Out(v)$ and $In(v)$ by forward/backward BFS from source v that is randomly picked from G. This process then repeats on $G[Out(v) - SCC(v)]$, $G[In(v) - SCC(v)]$

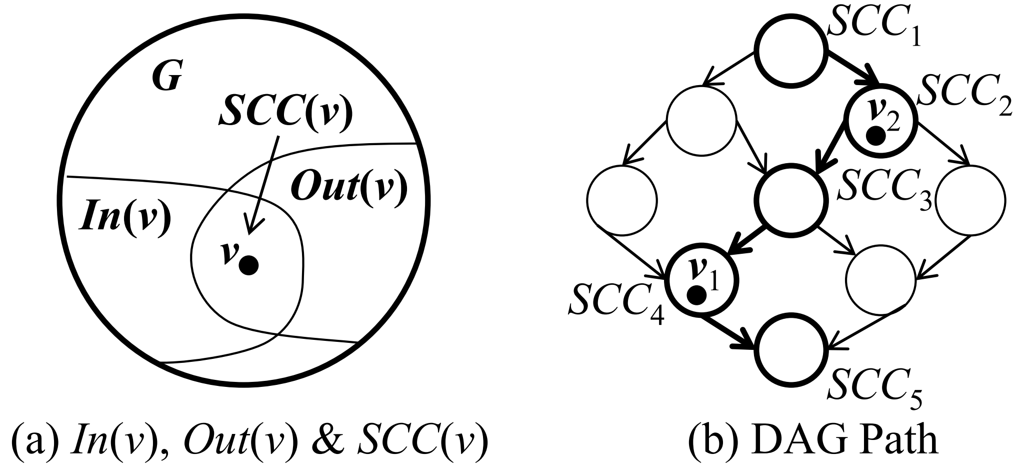


Figure 4.10: Concepts in SCC computation

and $G[V - (Out(v) \cup In(v))]$, where $G[X]$ denotes the subgraph of G induced by vertex set X. The correctness is guaranteed by the property that any remaining SCC must be in one of these subgraphs.

Our Contributions. It is difficult to translate the PRAM algorithms to work in Pregel. Thus, we design two Pregel algorithms based on label propagation. The first algorithm propagates the smallest vertex (ID) that every vertex has seen so far, while the second algorithm propagates multiple source vertices to speed up SCC computation. We also noticed a very recent algorithm that computes SCCs in Pregel [41], which shares a similar idea as our first algorithm. However, their algorithm performs label propagation for only one round, followed by a serial computation by the master machine, which is called FCS (Finishing Computations Serially). For processing large graphs, it is possible that the remaining graph after one round of computation is still too large to fit in the memory of a single machine. In contrast, we introduce graph decomposition, which allows us to run multiple rounds of label propagation. Moreover, our Optimization 2 described at the end of Section 4.5.1 performs a processing similar to FCS in [41], but it is fully distributed (by utilizing the property of recursive graph decomposition) instead of computation at the master machine. Finally, the idea used by our second SCC algorithm is new, which overcomes a weakness of our first algorithm.

Before describing our SCC algorithms, we first present a BPPA for graph decomposition which is used in our algorithms.

Graph Decomposition. Given a partition of V, denoted by V_1, V_2, ..., V_ℓ, we decompose G into $G[V_1]$, $G[V_2]$, ..., $G[V_\ell]$ in two supersteps (assume that each vertex v contains a label i indicating $v \in V_i$): (1)each vertex notifies all its in-neighbors and out-neighbors about its label i; (2)each vertex checks the incoming messages, removes the edges from/to the vertices whose label is different from its own label, and votes to halt.

4.5.1 Min-Label Algorithm

We first describe the Pregel operation for ***min-label*** propagation, where each vertex v maintains two labels $min_f(v)$ and $min_b(v)$.

Forward Min-Label Propagation. (1)Each vertex v initializes $min_f(v) = v$, propagates $min_f(v)$ to all v's out-neighbors, and votes to halt; (2)when a vertex v receives a set of messages from its in-neighbors $\Gamma_{in}(v)$, let min^* be the smallest message received, then if $min^* < min_f(v)$, v updates $min_f(v) = min^*$ and propagates $min_f(v)$ to all v's out-neighbors; v votes to halt at last. We repeat Step (2) until all vertices vote to halt.

Backward Min-Label Propagation. This operation is done after forward min-label propagation. The differences are that (1)initially, only vertices v satisfying $v = min_f(v)$ are active with $min_b(v) = v$, while for the other vertices u, $min_b(u) = \infty$; (2)each active vertex v propagates $min_b(v)$ towards all v's in-neighbors.

Both operations are BPPAs with $O(\delta)$ supersteps. After the forward and then backward min-label propagations, each vertex v obtains a label pair $(min_f(v), min_b(v))$. This labeling has the following property.

Lemma 4.5.1. *Let* $V_{(i,j)} = \{v \in V : (min_f(v), min_b(v)) = (i, j)\}$. *Then, (i)any SCC is a subset of some* $V_{(i,j)}$, *and (ii)*$V_{(i,i)}$ *is a SCC with color* i.

Proof. We first prove (i). Given a SCC and a vertex v in the SCC, suppose that v has a label pair (i, j), we show that for any other vertex u in the SCC, u also has the same label pair (i, j). We only prove $min_f(u) = i$, and the proof of $min_b(u) = j$ is symmetric. Let us denote $v_1 \to v_2$ iff v_1 can reach v_2. Since $i \to v \to u$, $min_f(u) > i$ is impossible. Also, since $min_f(u) \to u \to v$, $min_f(u) < i$ is impossible. Therefore, we have $min_f(u) = i$.

We now prove (ii). First, $\forall v \in V_{(i,i)}$, $v \to i$ and $i \to v$, and thus $V_{(i,i)} \subseteq SCC(i)$. Second, if $V_{(i,i)}$ exists, then $i \in V_{(i,i)}$, and by (i), we have $SCC(i) \subseteq V_{(i,i)}$. Therefore, $V_{(i,i)} = SCC(i)$. □

The **min-label algorithm** repeats the following operations: (1)forward min-label propagation; (2)backward min-label propagation; (3)an aggregator collects label pairs (i, j), and assigns a unique id $\mathcal{ID}$ to each $V_{(i,j)}$; then graph decomposition is performed to remove edges crossing different $G[V_{\mathcal{ID}}]$; finally, we mark each vertex v with label (i, i) to indicate that its SCC is found.

In each step, only unmarked vertices are active, and thus vertices do not participate in later rounds once its SCC is determined.

Each round of the algorithm refines the vertex partition of the previous round. Since all the three steps are BPPAs, each round of the min-label algorithm is a BPPA. The algorithm terminates once all vertices are marked.

The correctness of the algorithm follows directly from Lemma 4.5.1. We now analyze the number of rounds the min-label algorithm requires. Given a graph G, if we contract each SCC into a super-vertex, we obtain a DAG in which an edge directs from one super-vertex (representing a SCC, SCC_i) to another super-vertex (representing another SCC, SCC_j) iff there is an edge from some vertex in SCC_i to some vertex in SCC_j. Let $\mathcal{L}$ be the longest path length in the DAG, then we have the following bound.

Theorem 4.5.1. *The min-label algorithm runs for at most $\mathcal{L}$ rounds.*

Proof. We show the correctness of Theorem 4.5.1 by proving the following invariant: at the end of Round i, the length of any maximal path in the DAG is at most $\mathcal{L} - i$. We only need to show that in each round, a SCC is determined for any maximal path in the DAG. This is because, after the SCC is marked and the exterior edges are removed, (1) if the SCC is an endpoint of the DAG path (let us denote its length by ℓ), the path length becomes $\ell - 1$; (2) otherwise, the path is broken into two paths with length $< \ell - 1$. In either case, the new paths have length at most $\ell - 1$.

We now show that for an arbitrary maximal DAG path p, at least one SCC is found as $V_{(i,i)}$. First, let us assume i_1 is the smallest unmarked vertex in G, then $V_{(i_1,i_1)}$ is found as a SCC for any maximal path p_1 that contains $SCC(i_1)$. Now consider those maximal paths that does not contain $SCC(i_1)$, and let i_2 be the smallest vertex in the SCCs of these paths. Then $V_{(i_2,i_2)}$ is found as a SCC for any such path p_2 that contains $SCC(i_2)$. The reasoning continues for those maximal paths that do not contain $SCC(i_1)$ and $SCC(i_2)$ until all maximal paths are covered. □

The above bound is very loose, and often, more than one SCC is marked per DAG path in a round. We illustrate it using Figure 4.10(b), where there is a DAG path $P = \langle SCC_1, SCC_2, SCC_3, SCC_4, SCC_5\rangle$, and v_1 is the smallest vertex in G. Obviously, for any vertex v in SCC_4–SCC_5, $min_f(v) = v_1$. Since v_1 is picked as a source for backward propagation, for any vertex v in SCC_1–SCC_4, $min_b(v) = v_1$. Thus, any vertex $v \in SCC_4$ has label pair (v_1, v_1) and is marked. Now consider the sub-path before SCC_4, i.e., $\langle SCC_1$, $SCC_2, SCC_3\rangle$, and assume $v_2 \in SCC_2$ is the second smallest vertex in G, then a similar reasoning shows that SCC_2 is found as $V_{(v_2,v_2)}$. In this way, P is quickly broken into many subpaths, each can be processed in parallel, and hence in practice the number of rounds needed can be much less than $\mathcal{L}$.

In our implementation, we further perform two optimizations:

Optimization 1: Removing Trivial SCCs. If the in-degree or out-degree of a vertex v is 0, then v itself constitutes a trivial SCC and can be directly marked to avoid useless label propagation. We mark trivial SCCs before forward min-label propagation in each round.

We now describe a Pregel algorithm to mark all vertices with in-degree 0. Initially, each vertex v with in-degree 0 marks itself, sends itself to all out-neighbors and votes to halt. In subsequent supersteps, each vertex v removes its in-edges from the in-neighbors that appears in the incoming messages, and checks whether its in-degree is 0. If so, the vertex marks itself and sends itself to all out-neighbors. Finally, the vertex votes to halt. We also mark vertices with out-degree 0 in each superstep in a symmetric manner.

The algorithm takes only a small number of supersteps in practice (as shown by experiments in Section 4.6), since real world graphs (e.g., social networks) have a dense core, and the limited number of trivial SCCs only exist in the sparse boundary regions of the graphs. Furthermore, many vertices with zero in-degree/out-degree are marked as trivial SCCs in parallel in each superstep. On the other hand, removing trivial SCCs prevents them from participating in min-label propagation, which may otherwise degrade the algorithm effectiveness if some trivial SCC vertex has a small ID.

Optimization 2: Early Termination. We do not need to run the algorithm until all vertices are marked as a vertex of a SCC found. We also mark a vertex $v \in G[V_{\mathcal{ID}}]$ if $|V_{\mathcal{ID}}|$ is smaller than a threshold τ, so that all vertices in subgraph $G[V_{\mathcal{ID}}]$ remain inactive in later rounds. Here, $|V_{\mathcal{ID}}|$ is obtained using the aggregator of Step (3). We stop once all vertices are marked as a SCC/subgraph vertex. Then, we use one round of MapReduce to assign the subgraphs to different machines to compute the SCCs directly from each subgraph $G[V_{\mathcal{ID}}]$. Since each subgraph is small, its SCCs can be computed on a single machine using an efficient main memory algorithm, without inter-machine communication.

4.5.2 Multi-Label Algorithm

A real world graph usually has a giant SCC that contains the majority of its vertices, and it is desirable to find the giant SCC early (e.g., in the first round) so that we can terminate earlier by applying Optimization 2 mentioned above. However, the min-label algorithm may not find the giant SCC in the first round. For example, let the giant SCC be SCC_{max}, then if there is a vertex $v \notin SCC_{max}$ whose ID is smaller than all vertices in SCC_{max}, and if v links to a vertex in SCC_{max}, then SCC_{max} cannot be found in Round 1 by the min-label algorithm. On the other hand, the multi-label algorithm to be presented in this subsection almost always finds the giant SCC in the first round.

The multi-label algorithm aims to speed up SCC discovery and graph decomposition by propagating k source vertices in parallel, instead of just one randomly picked source as done by the min-label algorithm and existing algorithms [15, 5, 4, 41].

In this algorithm, each vertex v maintains two label sets $Src_f(u)$ and $Src_b(u)$. The algorithm is similar to the min-label algorithm, except that the min-label propagation operation is replaced with the *k-label* propagation operation described below:

Forward k-Label Propagation. Suppose that the current vertex partition is V_1, V_2, . . ., V_ℓ. (1)In Superstep 1, an aggregator randomly selects k vertex samples from each subgraph $G[V_i]$. (2)In Superstep 2, each source u initializes $Src_f(u) = \{u\}$ and propagates label u to all its out-neighbors, while each non-source vertex v initializes $Src_f(v) = \emptyset$. Finally, the vertex votes to halt. (3)In subsequent supersteps, if a vertex v receives a label $u \notin Src_f(v)$ from an in-neighbor, it updates $Src_f(v) = Src_f(v) \cup \{u\}$ and propagates u to all its out-neighbors, before voting to halt.

The backward k-label propagation is symmetric. Unlike the min-label algorithm where backward propagation is done after forward propagation, in the multi-label algorithm, we perform both forward and backward propagation in parallel.

The k-label propagation operation is also a BPPA with $O(\delta)$ supersteps, and when it terminates, each vertex v obtains a label pair $(Src_f(v), Src_b(v))$. This labeling has the following property.

Lemma 4.5.2. *Let* $V_{(S_f,S_b)} = \{v \in V : (Src_f(v), Src_b(v)) = (S_f, S_b)\}$. *Then, (i)any SCC is a subset of some* $V_{(S_f,S_b)}$, *and (ii)*$V_{(S_f,S_b)}$ *is a SCC if* $S_f \cap S_b \neq \emptyset$.

Proof. We first prove (i). Given a SCC and a vertex v in the SCC, suppose that v has a label pair (S_f, S_b), we show that for any other u in the SCC, u also has the same label pair (S_f, S_b). We only prove $Src_f(u) = S_f$, and the proof of $Src_b(u) = S_b$ is symmetric. (1) For any vertex $s \in S_f$, we have $s \to v \to u$ and thus $s \in Src_f(u)$; therefore, $S_f \subseteq Src_f(u)$. (2) For any vertex $s \in Src_f(u)$, we have $s \to u \to v$ and thus $s \in Src_f(v) = S_f$; therefore, $Src_f(u) \subseteq S_f$. As a result, $Src_f(u) = S_f$.

We now prove (ii), Since $S_f \cap S_b \neq \emptyset$, $\exists u \in S_f \cap S_b$. First, $\forall v \in V_{(S_f,S_b)}$, $v \to u$ and $u \to v$, and thus $V_{(S_f,S_b)} \subseteq SCC(u)$. Second, if $V_{(S_f,S_b)}$ exists, then $u \in V_{(S_f,S_b)}$ and by (i), we have $SCC(u) \subseteq V_{(S_f,S_b)}$. Therefore, $V_{(S_f,S_b)} = SCC(u)$. □

We now analyze the number of rounds required. In any round, we have ℓ subgraphs and thus around ℓk source vertices. Since we do not know ℓ, We only give a very loose analysis assuming $\ell = 1$ (i.e., there are only k sources). Furthermore, we assume that vertices are only marked because they form a SCC, while in practice Optimization 2 of Section 4.5.1 is applied to also mark vertices of sufficiently small subgraphs.

Suppose that we can mark $(1 - \theta)n$ vertices as SCC vertices in each round, where $0 < \theta < 1$. Then, after i rounds the graph has $\theta^i n$ vertices, and in $O(\log_{1/\theta} n)$ rounds the graph is sufficiently small to allow efficient single-machine SCC computation. We now study the relationship between θ and k.

Assume that there are c SCCs in G: $SCC_1,\ SCC_2, \ldots, SCC_c$. Let n_i be the number of vertices in SCC_i and $p_i = n_i/n$. We analyze how many vertices are marked in expectation after one round. Note that if x sampled source vertices belong to the same SCC, then we actually waste $x-1$ samples. Our goal is to show that such waste is limited.

We define a random variable X that refers to the number of vertices marked. We also define an indicator variable X_i for each SCC SCC_i as follows: $X_i = 1$ if at least one sample belongs to SCC_i, and $X_i = 0$ otherwise. Let s_j be the j-th sample. We have

$$\begin{aligned} E[X_i] &= Pr\{X_i = 1\} &= 1 - \prod_{j=1}^{k} Pr\{s_j \notin SCC_i\} \\ &= 1 - \prod_{j=1}^{k}(1-p_i) &= 1-(1-p_i)^k. \end{aligned}$$

Note that $X = \sum_{i=1}^{c} n_i \cdot X_i$. According to the linearity of expectation, we have

$$\begin{aligned} E[X] &= \sum_{i=1}^{c} n_i \cdot E[X_i] &= \sum_{i=1}^{c}[n_i - n_i(1-p_i)^k] \\ &= n - \sum_{i=1}^{c} n_i(1-p_i)^k &= n - n\sum_{i=1}^{c} p_i(1-p_i)^k. \end{aligned}$$

In other words, $\theta = \sum_{i=1}^{c} p_i(1-p_i)^k$. Since the number of vertices remaining unmarked is θn, we want θ to be as small as possible. In fact, if the size of SCCs are biased, θ is small. This is because if there is a very large SCC, it is likely that some of its vertices are sampled as source vertices, and hence many vertices will be marked as being a vertex of the SCC.

The worst case happens when all the SCCs are of equal size, i.e., $p_i = 1/c$ for all i, in which case $\theta = (1-1/c)^k$. Since $(1-1/c) < 1$, θ decreases with k, but the rate of decrement depends on c. For example, when $c = 1000$, to get $\theta = 0.9$ we need to set $k = 100$. However, we note that real world graphs rarely have all SCCs with similar sizes, and the analysis is very loose. In practice, k can be much smaller even for very large c.

We now present a theorem that formalizes the above discussion.

Theorem 4.5.2. *If $p_i < \frac{2}{k+1}$ for all i, then $\theta \leq (1-1/c)^k$. Otherwise, $\theta = \sum_{i=1}^{c} p_i(1-p_i)^k < 1 - 1/k$.*

Proof. We first prove the case when there exists a SCC SCC_i with $p_i \geq \frac{2}{k+1} > \frac{1}{k}$. Since we sample k vertices in total, in expectation at least one vertex in SCC_i is sampled. As a result, in expectation at least $n_i > n/k$ vertices are marked, or equivalently, $\theta < 1 - 1/k$.

We now prove the case when $p_i < \frac{2}{k+1}$ for all i. Consider the following optimization problem:

$$\begin{array}{ll} \text{maximize} & \theta(p_1, \ldots, p_c) = \sum_{i=1}^{c} p_i(1-p_i)^k \\ \text{subject to} & \sum_{i=1}^{c} p_i = 1,\ (p_i > 0) \end{array}$$

Using the method of Lagrange multipliers, we obtain the following Lagrange function:

$$L(p_1, \ldots, p_c, \lambda) = \sum_{i=1}^{c} p_i(1-p_i)^k + \lambda(\sum_{i=1}^{c} p_i - 1)$$

The stationary points can be obtained by solving the following equations:

$$\begin{aligned} \frac{\partial L}{\partial p_i} &= [1-(k+1)p_i](1-p_i)^{k-1} + \lambda \triangleq 0 \\ \frac{\partial L}{\partial \lambda} &= \sum_{i=1}^{c} p_i - 1 \triangleq 0 \end{aligned}$$

Obviously, $p_i = 1/c$ and $\lambda = [(k+1)/c - 1](1-1/c)^{k-1}$ is a solution. To prove that θ is maximized at this point (i.e., $p_i = 1/c$ for all i), we need to show that θ is a concave function in the domain $\{(p_1, \ldots, p_c) \mid p_i > 0 \text{ for all } i \text{ and } \sum_{i=1}^{c} p_i = 1\}$. This is equivalent to showing that the hessian matrix of θ is negative definite.

We now compute the elements of the hessian matrix:

$$\begin{aligned} \frac{\partial^2 \theta}{\partial p_i \partial p_j} &= 0 \qquad\qquad \text{(when } i \neq j\text{)} \\ \frac{\partial^2 \theta}{\partial p_i^2} &= k[(k+1)p_i - 2](1-p_i)^{k-2} \end{aligned}$$

Therefore, the hessian matrix is a diagonal matrix $diag(\frac{\partial^2 \theta}{\partial p_1^2}, \cdots, \frac{\partial^2 \theta}{\partial p_c^2})$. We now show that it is negative definite, which is based on the following property from linear algebra:

Lemma 4.5.3. *Matrix $M_{N\times N}$ is negative definite iff for all $r = 1, \cdots, N$, $(-1)^r \det({}_rM_r) > 0$, where ${}_sM_t$ is the submatrix composed of the first t rows and s columns of M.*

The proof is completed by observing that

$$\begin{aligned} & (-1)^r \det({}_rM_r) \\ = \;& (-1)^r \det(diag(\frac{\partial^2\theta}{\partial p_1^2}, \cdots, \frac{\partial^2\theta}{\partial p_r^2})) \\ = \;& (-1)^r \prod_{i=1}^{r} \frac{\partial^2\theta}{\partial p_i^2} \\ = \;& k^r \cdot [\prod_{i=1}^{r}(2-(k+1)p_i))] \cdot [\prod_{i=1}^{r}(1-p_i)]^{k-2} \\ > \;& 0 \qquad \text{(since } p_i < 2/(k+1) \text{ and } p_i < 1) \end{aligned}$$

□

Finally, we emphasize that Theorem 4.5.2 is very loose: when there exists a SCC SCC_i with p_i much greater than $\frac{2}{k+1}$, θ is much smaller than $1 - 1/k$.

4.6 Experimental Evaluation

We evaluate the performance of our algorithms over large real-world graphs. We ran all experiments on a cluster of 16 machines, each with 24 processors (two Intel Xeon E5-2620 CPU) and 48GB RAM. One machine is used as the master that runs only one working process (or simply, worker), while the other 15 machines act as slaves each running 10 workers. The connectivity between any pair of nodes in the cluster is 1Gbps.

All our algorithms were implemented in Pregel+[1], which is an open-source implementation of Pregel, though any Pregel-like system can be used to implement our algorithms. We remark that we did not use any optimization techniques in Pregel+, i.e., we used only the basic features of Pregel, as our aim is to test the performance of our algorithms in a general Pregel-like system. All the source codes of the algorithms discussed in this chapter can be found at `http://www.cse.cuhk.edu.hk/pregelplus/download.html`.

Datasets. We used 10 real-world graph datasets, which are listed in Figure 4.11: (1)*BTC*[2]: a semantic graph converted from the Billion Triple Challenge 2009 RDF dataset [9];

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

[2] http://km.aifb.kit.edu/projects/btc-2009/

Data	**Type**	$\|V\|$	$\|E\|$
BTC	undirected	164,732,473	772,822,094
LJ-UG	undirected	10,690,276	224,614,770
Facebook	undirected	59,216,214	185,044,032
USA	undirected	23,947,347	58,333,344
Euro	undirected	18,029,721	44,826,904
Twitter	directed	52,579,682	1,963,263,821
LJ-DG	directed	4,847,571	68,993,773
Pokec	directed	1,632,803	30,622,564
Flickr	directed	2,302,925	33,140,017
Patent	directed	3,774,768	16,518,948

Figure 4.11: Datasets

(2)*LJ-UG*[3]: a network of LiveJournal users and their group memberships; (3)*Facebook*[4]: a friendship network of the Facebook social network; (4)*USA*[5]: the USA road network; (5)*Euro*[6]: the European road network; (6)*Twitter*[7]: Twitter who-follows-who network based on a snapshot taken in 2009; (7)*LJ-DG*[8]: a friendship network of the LiveJournal blogging community; (8)*Pokec*[9]: a friendship network of the Pokec social network; (9)*Flickr*[10]: a friendship network of the Flickr social network; (10)*Patent*[11]: the US patent citation network.

4.6.1 Performance Comparison with GraphLab

As discussed in Section 4.1, GraphLab [30] (and PowerGraph [17]) only allows a vertex to access the states of its adjacent vertices and edges. As a result, it does not support algorithms in which a vertex needs to communicate with a non-neighbor, such as the S-V algorithm in Section 4.3.2, the list ranking algorithm in Section 4.4.2 and the algorithm presented in Section 4.4.2. Thus, we cannot implement our BCC algorithm in GraphLab. Moreover, we also cannot implement our SCC algorithms in GraphLab, because GraphLab

[3] http://konect.uni-koblenz.de/networks/livejournal-groupmemberships

[4] http://konect.uni-koblenz.de/networks/facebook-sg

[5] http://www.dis.uniroma1.it/challenge9/download.shtml

[6] http://www.dis.uniroma1.it/challenge9/download.shtml

[7] http://konect.uni-koblenz.de/networks/twitter_mpi

[8] http://snap.stanford.edu/data/soc-LiveJournal1.html

[9] http://snap.stanford.edu/data/soc-pokec.html

[10] http://konect.uni-koblenz.de/networks/flickr-growth

[11] http://snap.stanford.edu/data/cit-Patents.html

	BTC			USA		
	Pregel+	GraphLab		Pregel+	GraphLab	
		Sync	Async		Sync	Async
# of Supersteps	30	30	N/A	6262	6262	N/A
Computing Time (sec)	32.24	83.1	155	1011	2982	627

Figure 4.12: Pregel+ and GraphLab running Hash-Min

does not support graph mutations, while the graph decomposition operation in our SCC algorithms involves edge deletion.

Due to the above-mentioned limitations, we only compare the performance of GraphLab with Pregel+ for the Hash-Min algorithm. We use GraphLab 2.2, which includes all the features of PowerGraph [17], and ran both GraphLab's asynchronous and synchronous modes. GraphLab's synchronous mode simulates Pregel, but due to the above-mentioned limitations, it is also difficult to implement the S-V, BCC and SCC algorithms in its synchronous mode.

Figure 4.12 reports the performance of Pregel+ and GraphLab when running Hash-Min over the small-diameter *BTC* graph (with skewed degree distribution) and the large-diameter *USA* road network (in which the degree of all vertices is small). The result shows that Pregel+ is significantly faster than GraphLab for processing the small-diameter *BTC* graph. For the large-diameter *USA* graph, Pregel+ is almost 3 times faster than the synchronous GraphLab, but is 1.6 times slower than the asynchronous GraphLab (for the reason given below).

For a large-diameter graph like *USA*, asynchronous execution is faster than its synchronous mode. This is because in asynchronous execution, the update to $min(v)$ is immediately visible to all other vertices, while in synchronous execution, the update is visible to other vertices only at the next superstep, resulting in a slower convergence. However, for a small-diameter graph like *BTC*, the synchronous execution converges in only 30 supersteps; thus, even though the asynchronous mode can converge faster, the gain is not big enough to cover the overhead of data locking/unlocking in asynchronous execution.

Overall, the much superior performance of Pregel+ on small-diameter graphs with skewed degree distribution, and the reasonable performance of Pregel+ on large-diameter graphs, justify that Pregel+ is a good choice of distributed graph computing system for implementing our algorithms. In addition, the limitations of GraphLab discussed above make the implementation of certain categories of graph algorithms difficult using GraphLab, which further justifies the adoption of Pregel+ in our work.

Finally, we remark that this chapter focuses on practical algorithms for Pregel-like systems rather than on the systems themselves, and we refer readers to our online report on a comprehensive comparison of existing systems including GraphLab, Giraph [1], GPS [40], and Pregel+ [55].

4.6.2 Performance of CC & BCC Algorithms

In Section 4.4 we proposed PPAs/BPPAs for a list of fundamental graph problems. Since they are used as building blocks in the PPA for computing BCCs, we also report their performance results as the steps of the BCC computation. Recall from Section 4.4.2 that the sequence of PPA tasks in the BCC computation include: (1)-(2)either *HashMin* + *BFS*, or *S-V*; (3)*EulerTour*; (4)*ListRank1*; (5)*ListRank2*; (6)*MinMax*; (7)*AuxGraph*; (8)either *HashMin2* or *S-V2*; (9)*Case1Mark*. Among them, Tasks (1) and (8) also report the performance of our two PPAs for computing CCs described in Section 4.3.

We report the per-task performance of BCC computation on the three *small-diameter* graphs *BTC*, *LJ-UG* and *Facebook* in Figure 4.13. Due to the small graph diameter, Hash-Min is much more efficient than S-V over these graphs. For example, Hash-Min finishes in 18 supersteps on *LJ-UG* and uses only 11.85 seconds. In contrast, S-V takes 58 supersteps and 142.24 seconds. Thus, the results verify that it is more efficient to compute CCs using Hash-Min when the graph diameter is small. We also give the total computational time of our BCC algorithm, and the results again show that using Hash-Min as a building block in the BCC computation achieves almost twice shorter total time than using S-V.

Next, we report the per-task performance of BCC computation on the two *large-diameter* road networks *USA* and *Euro* in Figure 4.14. Due to the large graph diameter, Hash-Min is very time-consuming. For example, it takes 1011.19 seconds and 6262 supersteps on *USA*. In contrast, S-V takes only 198 supersteps and 368.20 seconds. This again shows the difference between an $O(\delta)$-superstep PPA (e.g., Hash-Min) and an $O(\log n)$-superstep PPA (e.g., S-V). Similar behavior is also observed for CC computation over G^*, where HashMin2 uses 5437.72 seconds on *USA* while S-V2 on only 526.69 seconds. Overall, the S-V based BCC algorithm is 5.85 times faster than the Hash-Min based BCC algorithm on *USA*, and 4.17 times faster on *Euro*. This demonstrates the advantage of our S-V algorithm for processing large-diameter graphs.

It might be argued that computing the CCs of a road network in Pregel is not important, as road networks are usually connected and not very large. However, some spatial networks are huge in size, such as the triangulated irregular network (TIN) that models terrain, where CC computation is useful when we want to compute the islands given a specific sea level. Also, CC computation is a critical building block in our PPA for computing BCCs, and finding BCCs of a spatial network is important for analyzing its weak connection points.

Task	**BTC**		**LJ-UG**		**Facebook**	
	# of Steps	**Comp. Time**	**# of Steps**	**Comp. Time**	**# of Steps**	**Comp. Time**
HashMin	30	32.24 s	18	11.85 s	16	37.10 s
BFS	31	20.56 s	19	8.61 s	17	10.14 s
S-V	86	449.97 s	58	142.24 s	72	337.02 s
EulerTour	3	14.26 s	3	2.26 s	3	7.56 s
ListRank1	49	544.71 s	53	97.98 s	57	408.79 s
ListRank2	49	541.86 s	53	98.59 s	57	411.17 s
MinMax	46	35.88 s	50	8.54 s	54	20.42 s
AuxGraph	4	58.05 s	4	21.94 s	4	22.52 s
HashMin2	34	42.91 s	11	21.04 s	16	43.31 s
S-V2	72	443.16 s	58	138.59 s	86	385.86 s
Case1Mark	4	35.62 s	4	20.83 s	4	13.27 s
Total Time (CC by HashMin)	1326.09 s		291.64 s		974.28 s	
Total Time (CC by S-V)	2123.51 s		530.97 s		1606.61 s	

Figure 4.13: CC/BCC performance on BTC, LJ-UG, & Facebook

4.6.3 Performance of SCC Algorithms

We now report the performance of our min-label and multi-label algorithms for computing SCCs on the directed graphs.

Performance of Min-Label Algorithm

Before describing the results, we first review the sequence of tasks performed in each round of our min-label algorithm: (1)*Opt 1*: this task removes trivial SCCs as described in Optimization 1 of Section 4.5.1; (2)*MinLabel*: forward min-label propagation followed by backward min-label propagation; (3)*GDecom*: this task uses an aggregator to collect label pairs $(min_f(u), min_b(u))$ and assigns a new $\mathcal{ID}$ to each pair, sets the $\mathcal{ID}$ of each vertex u according to u's $(min_f(u), min_b(u))$, marks each vertex u with $min_f(u) = min_b(u)$ as being in a SCC, and performs graph decomposition using the algorithm described at the beginning of Section 4.5. Recall that we do not decompose a subgraph if its size (decided by number of vertices) is smaller than a user-defined threshold τ.

Task	**USA**		**Euro**	
	# of Steps	**Comp. Time**	**# of Steps**	**Comp. Time**
HashMin	6262	1011.19 s	4896	692.39 s
BFS	6263	964.11 s	4897	639.78 s
S-V	198	368.20 s	212	340.19 s
EulerTour	3	3.04 s	3	2.42 s
ListRank1	55	203.05 s	55	165.89 s
ListRank2	55	197.78 s	55	160.49 s
MinMax	52	16.65 s	52	12.77 s
AuxGraph	4	12.29 s	4	9.89 s
HashMin2	7365	5437.72 s	2836	2935.28 s
S-V2	226	536.69 s	184	414.20 s
Case1Mark	4	2.90 s	4	1.94 s
Total Time (CC by HashMin)	7848.73 s		4620.85 s	
Total Time (CC by S-V)	1340.60 s		1107.79 s	

Figure 4.14: CC/BCC performance on USA & Euro

We first compute the SCCs of the largest graph, *Twitter*, where we only mark a vertex when its SCC is determined (i.e., τ=0). Figure 4.15 reports the number of supersteps and the computational time taken by each task. The last column "Max Size" shows the maximum $|V_{\mathcal{ID}}|$ among those subgraphs, $G[V_{\mathcal{ID}}]$, that are not marked as a SCC after each round, and all SCCs are found when *MaxSize* becomes 0. As Figure 4.15 shows, the min-label algorithm takes only 4 rounds to compute all the SCCs over *Twitter*, which demonstrates that in practice the min-label algorithm requires much less than $\mathcal{L}$ rounds given in Theorem 4.5.1. Besides, the min-label propagation operations take only a small number of supersteps due to the small graph diameter. For example, in Round 1, forward propagation takes only 15 supersteps, followed by a 14-superstep backward propagation. The total computational time is only 187.94 seconds for a graph with almost 2 billion edges, which is very efficient.

Note that after Round 2 in Figure 4.15, the largest unmarked subgraph has size merely 22. Therefore, another option is to distribute these small subgraphs to different machines for single-machine SCC computation using MapReduce. Thus, we also run our min-label algorithm over *Twitter* using $\tau = 50,000$, so that a subgraph is marked to avoid further decomposition once it contains less than 50,000 vertices. All vertices are marked after 2

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	18	9.41 s	238,986
	MinLabel	15 + 14	75.80 s	
	GDecom	3	76.86 s	
2	**Opt 1**	5	0.81 s	22
	MinLabel	36 + 75	18.73 s	
	GDecom	3	1.74 s	
3	**Opt 1**	3	0.46 s	2
	MinLabel	6 + 5	2.02 s	
	GDecom	3	0.44 s	
4	**Opt 1**	1	0.21 s	0
	MinLabel	3 + 3	0.96 s	
	GDecom	3	0.50 s	
Total			187.94 s	

Figure 4.15: Min-label performance on Twitter ($\tau = 0$)

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	16	2.73 s	51,697
	MinLabel	14 + 16	14.91 s	
	GDecom	3	7.24 s	
2	**Opt 1**	4	0.34 s	81
	MinLabel	8 + 9	1.91 s	
	GDecom	3	0.72 s	
3	**Opt 1**	3	0.22 s	29
	MinLabel	6 + 7	1.08 s	
	GDecom	3	0.40 s	
4	**Opt 1**	1	0.14 s	12
	MinLabel	4 + 4	1.01 s	
	GDecom	3	0.32 s	
Total			31.02 s	

Figure 4.16: Min-label performance on LJ-DG ($\tau = 0$)

rounds, and the performance is similar to those in Figure 4.15. We then run a MapReduce job to compute the SCCs of the marked subgraphs, which takes 199 seconds.

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	6	0.47 s	2,509
	MinLabel	13 + 13	9.13 s	
	GDecom	3	3.26 s	
2	**Opt 1**	24	2.66 s	8
	MinLabel	6 + 5	1.37 s	
	GDecom	3	0.31 s	
3	**Opt 1**	1	0.10 s	0
	MinLabel	6 + 4	0.53 s	
	GDecom	3	0.26 s	
Total			18.09 s	

Figure 4.17: Min-label performance on Pokec ($\tau = 0$)

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	5	0.38 s	124,164
	MinLabel	16 + 18	7.23 s	
	GDecom	3	3.51 s	
2	**Opt 1**	4	0.35 s	81
	MinLabel	12 + 9	1.79 s	
	GDecom	3	0.81 s	
3	**Opt 1**	2	0.27 s	35
	MinLabel	7 + 8	1.74 s	
	GDecom	3	0.31 s	
4	**Opt 1**	1	0.09 s	0
	MinLabel	5 + 5	1.17 s	
	GDecom	3	0.23 s	
Total			17.88 s	

Figure 4.18: Min-label performance on Flickr ($\tau = 0$)

For the three relatively smaller graphs, *Pokec*, *Flickr* and *Patent*, the min-label algorithm with $\tau = 0$ finds all the SCCs in less than 4 rounds, and uses 18.09, 17.88, and 2.34 seconds, respectively. As shown in Figures 4.17, 4.18 and 4.19, the min-label algorithm with $\tau = 0$ finds all the SCCs of *Pokec*, *Flickr* and *Patent* in 3, 4 and 1 round(s), respec-

Round	**Task**	**# of Steps**	**Comp. Time**	**Max Size**
1	**Opt 1**	17	1.77 s	0
	MinLabel	2 + 2	0.28 s	
	GDecom	3	0.29 s	
Total			2.34 s	

Figure 4.19: Min-label performance on Patent ($\tau = 0$)

Round	**Task**	**# of Steps**	**Comp. Time**	**Max Size**
1	**Opt 1**	18	8.07 s	238,986
	MultiLabel	17	423.85 s	
	GDecom	3	102.98 s	
2	**Opt 1**	5	0.81 s	206,319
	MultiLabel	5	0.55 s	
	GDecom	3	0.41 s	
3	**Opt 1**	1	0.16 s	206,292
	MultiLabel	6	0.94 s	
	GDecom	3	0.38 s	
MapReduce			181 s	

Figure 4.20: Multi-label performance on Twitter ($\tau = 50,000$)

tively. If we run the min-label algorithm with $\tau = 50,000$ for two rounds, and then run a MapReduce job to compute the SCCs of the marked subgraphs, the MapReduce job takes 24 seconds on *Pokec* and 25 seconds on *Flickr*.

However, we remark that the min-label algorithm with $\tau = 0$ is not always able to find all the SCCs for an arbitrary graph. One example is the *LJ-DG* datasets, the performance of which is shown in Figure 4.16 (only the results of the first 4 rounds are shown). In fact, for the subsequent three rounds, “Max Size” decreases slowly as 11, 10 and 9. On the contrary, running min-label algorithm on *LJ-DG* using $\tau = 50,000$ takes only 2 rounds to mark all vertices, followed by a MapReduce job that computes the SCCs of the marked subgraphs in 27 seconds.

Round	Task	# of Steps	Comp. Time	Max Size
1	**Opt 1**	16	2.66 s	51,697
	MultiLabel	19	27.02 s	
	GDecom	3	6.30 s	
2	**Opt 1**	5	0.74 s	50,706
	MultiLabel	6	0.71 s	
	GDecom	3	0.25 s	
3	**Opt 1**	1	0.13 s	50,629
	MultiLabel	8	0.80 s	
	GDecom	3	0.36 s	
MapReduce			26 s	

Figure 4.21: Multi-label performance on LJ-DG ($\tau = 50,000$)

Performance of Multi-Label Algorithm

We now report the performance of our multi-label algorithm. For the parallel forward and backward k-label propagation, we fix $k = 10$. We also set $\tau = 50,000$ and subgraphs with less than $50,000$ vertices are not further decomposed. The performance of the multi-label algorithm on the two larger graphs, *Twitter* and *LJ-DG*, are shown in Figures 4.20 and 4.21. We can see that although Round 1 bounds the maximum unmarked subgraph size to a relatively small number, "Max Size" decreases slowly in the later rounds and we cannot afford to run till it gets smaller than $50,000$. However, the subgraphs are small enough to be assigned to different machines for local SCC computation using MapReduce, and the final round of MapReduce postprocessing is efficient for both graphs.

We now report the performance of our multi-label algorithm. The performance of the multi-label algorithm on *Pokec*, *Flickr* and *Patent* are shown in Figures 4.22–4.24. From Figure 4.23, we can see that although Round 1 bounds the maximum unmarked subgraph size to a relatively small number, "Max Size" decreases slowly in the later rounds and we cannot afford to run till it gets smaller than $50,000$. However, the subgraphs are small enough to be assigned to different machines for local SCC computation using MapReduce. On the other hand, from Figures 4.22 and 4.24, we can see that the multi-label algorithm performs well on *Pokec* and *Patent*, since there is no subgraph with at least $50,000$ vertices after Round 1. Finally, we remark that the final round of MapReduce postprocessing is efficient on all the three graphs.

Unlike the min-label algorithm for which we can afford to run until termination, the multi-label algorithm finds at most k SCCs in each round, and is only effective in ear-

Round	Task	# of Steps	Comp. Time	Max Size
1	Opt 1	6	0.70 s	0
	MultiLabel	15	15.98 s	
	GDecom	3	5.01 s	
MapReduce			24 s	

Figure 4.22: Multi-label performance on Pokec ($\tau = 50,000$)

Round	Task	# of Steps	Comp. Time	Max Size
1	Opt 1	5	0.57 s	125,528
	MultiLabel	21	19.30 s	
	GDecom	3	4.13 s	
2	Opt 1	4	0.33 s	125,443
	MultiLabel	9	0.47 s	
	GDecom	3	0.26 s	
3	Opt 1	1	0.09 s	125,336
	MultiLabel	10	1.09 s	
	GDecom	3	0.27 s	
MapReduce			74 s	

Figure 4.23: Multi-label performance on Flickr ($\tau = 50,000$)

Round	Task	# of Steps	Comp. Time	Max Size
1	Opt 1	17	2.71 s	0
	MultiLabel	3	0.19 s	
	GDecom	3	0.36 s	
MapReduce			24 s	

Figure 4.24: Multi-label performance on Patent ($\tau = 50,000$)

lier rounds when there are large SCCs. However, as discussed at the beginning of Section 4.5.2, the multi-label algorithm almost always finds the largest SCC in the first round, which is more desirable than the min-label algorithm. Thus, in applications where only the largest SCC (also called the giant SCC) is needed, the multi-label algorithm will be a better choice; in applications where all SCCs are needed, running multi-label algorithm for Round 1 followed by min-label algorithm for the subsequent rounds can be a good choice.

4.7 Conclusions

We proposed efficient distributed algorithms for computing three fundamental graph connectivity problems, namely CC, BCC, and SCC. Specifically, we defined the notion of PPA to design Pregel algorithms that have guaranteed performance, i.e., requiring only linear space, communication and computation per iteration, and only $O(\log n)$ or $O(\delta)$ iterations of computation. Experiments on large real-world graphs verified that our algorithms have good performance in shared-nothing parallel computing platforms.

Chapter 5

Extensive Performance Study

With the prevalence of graph data in real-world applications (e.g., social networks, mobile phone networks, web graphs, etc.) and their ever-increasing size, many distributed graph computing systems have been developed in recent years to process and analyze massive graphs. Most of these systems adopt Pregel's vertex-centric computing model, while various techniques have been proposed to address the limitations in the Pregel framework. However, there is a lack of comprehensive comparative analysis to evaluate the performance of various systems and their techniques, making it difficult for users to choose the best system for their applications. We conduct extensive experiments to evaluate the performance of existing systems on graphs with different characteristics and on algorithms with different design logic. We also study the effectiveness of various techniques adopted in existing systems, and the scalability of the systems. The results of our study reveal the strengths and limitations of existing systems, and provide valuable insights for users, researchers and system developers.

In this chapter, we first give a survey on existing distributed graph computing systems in Section 5.2. Among these systems, we conduct comprehensive experimental evaluation on Giraph [1], GraphLab/ PowerGraph [30, 16], GPS [40], Pregel+ [55], and GraphChi [27]. We do not conduct experiments on other existing systems for various reasons given in Section 5.2.7. In Section 5.3, we discuss a set of graph algorithms that are popularly used to evaluate the performance of various systems in existing works, and we classify them into five categories where each category represents a different logic of distributed algorithm design. Then, in Section 5.4, we conduct a comprehensive analysis on the performance of various systems focusing on the following key objectives:

- To evaluate the performance of various systems in processing large graphs with different characteristics, including skewed (e.g., power-law) degree distribution, small diameter (e.g., small-world), large diameter, (relatively) high average degree, and random graphs.

- To evaluate the performance of various systems with respect to different categories of algorithms presented in Section 5.3.

- To study the effects of individual techniques used in various systems on their performance, hence analyzing the strengths and limitations of each system. The techniques to be examined include message combiner, mirroring, dynamic repartitioning, request-respond API, asynchronous and synchronous computation, and support for graph mutation. We also study the effects of a few algorithmic optimizations [41] and compare with GraphChi [27] as a single machine baseline.

- To test the scalability of various systems by varying the number of machines and CPU cores, the number of vertices and edges in graphs with different degree distributions.

Related work. Guo et al. [18] proposed a benchmarking suite to compare the performance of various systems for distributed graph computation. Their benchmark defines a comprehensive evaluation process to select representative metrics, datasets and algorithm categories. While this is a pioneering work in benchmarking distributed graph computing systems, the authors also admit that their method has limitations in terms of selection of metrics and algorithmic coverage. Satish et al. [42] evaluated the performance of a number of systems, both distributed and single-machine, and identified the potential performance gap between these systems and the hand-optimized baseline code (whose performance is close to hardware limits). The results were then used to provide insights on how the systems may be improved to better utilize hardware capacity (e.g., memory/network bandwidth). Very recently (after the submission of our paper), we noticed the work by Han et al. [19], which evaluated four systems for their performance and useability, and identified potential areas of improvement in each system. Although both [19] and our work evaluated Giraph, GraphLab and GPS, due to different focus some findings are different. For example, the performance results of GPS we obtained are very different as we found that GPS's large fixed overhead can be easily eliminated by setting its polling time appropriately. We also evaluated a new system, Pregel+, which records better performance in many aspects, while [19] present results (e.g., memory usage, network I/O) that we do not report. Compared with [18] and [42], we focus on the performance evaluation of vertex-centric distributed graph computing systems, among which, only GraphLab and Giraph were evaluated in [18] and [42]. We also used more algorithms and among them, only three were used in [18], [19] and [42]. Besides large real graphs, we also used larger synthetic random and power-law graphs to analyze the scalability of the systems. We studied the effects of different system optimization techniques, as well as algorithmic optimizations [41], on the performance of the systems, which were not studied in [18], [19] and [42]. We analyzed in greater details the differences between GraphLab's asynchronous and synchronous modes. We also compared with GraphChi [27] as a single machine baseline. Thus, though [18], [19] and [42] made significant contributions in the evaluation of graph-computing systems, we believe that our work has also made substantial new contributions.

5.1 Preliminary

We first define some basic graph notations. Let $G = (V, E)$ be a graph, where V and E are the sets of vertices and edges of G. If G is undirected, we denote the set of neighbors of a vertex $v \in V$ by $\Gamma(v)$. If G is directed, we denote the set of in-neighbors (out-neighbors) of a vertex v by $\Gamma_{in}(v)$ ($\Gamma_{out}(v)$). Each vertex $v \in V$ has a unique integer ID, denoted by $id(v)$. The diameter of G is denoted by $\delta(G)$, or simply δ when G is clear from the context.

The distributed graph computing systems evaluated in this chapter are all based on a shared-nothing architecture, where data are stored in a *distributed file system (DFS)*, e.g., *Hadoop's DFS*. The input graph is stored as a distributed file in a DFS, where each line records a vertex and its adjacency list. A distributed graph computing system consists of a cluster of k workers, where each worker w_i keeps and processes a batch of vertices in its main memory. Here, "worker" is a general term for a computing unit, and a machine can have multiple workers in the form of threads/processes.

A job is processed by a graph computing system in three phases as follows. **(1)Loading**: each worker loads a portion of vertices from the DFS into its main-memory; then workers exchange vertices through the network (e.g., by hashing on vertex ID as in Pregel) so that each worker w_i finally keeps all and only those vertices that are assigned (i.e., hashed) to w_i. **(2)Iterative computing**: in each iteration, each worker processes its own portion of vertices sequentially, while different workers run in parallel and exchange messages. **(3)Dumping**: each worker writes the output of all its processed vertices to the DFS. Most existing graph-parallel systems follow the above three-phase procedure.

5.2 A Survey on Existing Systems

We briefly discuss existing distributed graph computing systems, and highlight their distinguished features.

5.2.1 Pregel

Pregel [33] is implemented in C/C++ and designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to different machines in a cluster, where each vertex v is associated with the set of v's neighbors. A program in Pregel implements a user-defined *compute*() function and proceeds in iterations (called *supersteps*). In each superstep, the program calls *compute*() for each active vertex. The *compute*() function performs the user-specified task for a vertex v, such as processing v's incoming messages (sent in the previous superstep), sending messages to other vertices (to be received in the next superstep), and making v vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Message Combiner. If x messages are to be sent from a machine M_i to a vertex v in a machine M_j, and some commutative and associative operation is to be applied to the x messages in *v.compute*() when they are received by v, then these x messages can be first combined into a single message which is then sent from M_i to v in M_j. To achieve this goal, Pregel allows users to implement a *combine*() function to specify how to combine messages that are sent from machine M_i to the same vertex v in machine M_j.

5.2.2 Giraph

Giraph is implemented in Java by Yahoo! as an open source of Google's Pregel. Later, Facebook built its Graph Search services upon Giraph, and further improved the performance of Giraph by introducing the following optimizations:

Multi-threading. Facebook adds multithreading to graph loading, dumping, and computation. In CPU bound applications, a speedup near-linear with the number of processors can be observed by multi-threading.

Memory optimization. The initial release of Giraph by Yahoo! requires high memory consumption, since all data types are stored as separate Java objects. A large number of Java objects greatly degrades the performance of Java Virtual Machine (JVM). Since object deletion is handled by Java's Garbage Collector (GC), if a machine maintains a large number of vertex/edge objects in main memory, GC needs to track a lot of objects and the overhead can severely degrade the system performance. To decrease the number of objects being maintained, Java-based systems maintain vertices in main memory in their binary representation. Thus, the later Giraph system organizes vertices and messages as main memory pages, where each page is simply a byte array object that holds the binary representation of many vertices.

5.2.3 GPS

GPS [40] is another open source Java implementation of Google's Pregel, with additional features. GPS extends the Pregel API to include an additional function, *master.compute*(), which provides the ability to access to all of the global aggregated values, and store global values which are transparent to the vertices. The global aggregated values can be updated before they are broadcast to the workers. Furthermore, GPS also introduces the following two techniques to boost system performance:

Large adjacency-list partitioning (LALP). When LALP is applied, adjacency lists of high-degree vertices are not stored in a single worker, but they are rather partitioned across workers. For each partition of the adjacency list of a high-degree vertex, a mirror of the vertex is created in the worker that keeps the partition. When a high-degree vertex broadcasts a message to its neighbors, at most one message is sent to its mirror at each machine. Then, the message is forwarded to all its neighbors in the partition of the adjacency list of the high-degree vertex.

Dynamic repartitioning (DP). DP repartitions the graph dynamically according to the workload during the execution process, in order to balance the workload among all workers and reduce the number of messages sent over the network. However, DP also introduces extra network workload to reassign vertices among workers, and the overhead can exceed the benefits gained.

5.2.4 Pregel+

Pregel+ [55] is implemented in C/C++ and each worker is simply an MPI process. In addition to the basic techniques provided in existing Pregel-like systems, Pregel+ introduces two new techniques to further reduce the number of messages.

Mirroring. Mirroring is similar to LALP in GPS; but Pregel+ integrates both mirroring and message combiner, and the Pregel+ system selects vertices for mirroring based on a cost model that analyzes the tradeoff between mirroring and message combining. Thus, the integration of mirroring and message combiner in Pregel+ leads to significantly more effective message reduction than applying combiner alone. Pregel+ also supports an additional API that allows mirroring to be used in applications where message values depend on the edge fields (e.g., single-source shortest path computation), which is not supported by LALP in GPS.

Request-Respond API. This API allows a vertex u to request another vertex v for a value, $a(v)$, and the requested value will be available to u in the next iteration. The technique can effectively reduce the number of messages passed, since all requests from a machine to the same target vertex v are merged into one request (e.g., as Figure 1.3 shows, requests from all u_i in machine M_1 are merged into one request sent to v_j in machine M_2).

5.2.5 GraphLab and PowerGraph

GraphLab 2.2. (which includes PowerGraph) is implemented in C/C++. Unlike Pregel's *synchronous data-pushing model* and *message passing paradigm*, GraphLab [30] adopts an *Gather, Apply, Scatter* (*GAS*) *data-pulling model* and shared memory abstraction. A program in GraphLab implements a user-defined *GAS* function for each vertex. To avoid the imbalanced workload caused by high-degree vertices in power-law graphs, a recent version of GraphLab, called PowerGraph [16], introduces a new graph partition scheme to handle the challenges of power-law graphs as follows.

In the *Gather* phase, each active vertex collects information from adjacent vertices and edges, and performs a generalized sum operation over them. This generalized sum operation must be commutative and associative, ranging from a numerical sum to the union of the collected information. In the *Apply* phase, each active vertex can update its value based on the result of the generalized sum and its old value. Finally, in the *Scatter* phase, each active vertex can activate the adjacent vertices. However, unlike Pregel's message passing paradigm, GraphLab can only gather information from adjacent edges and scatter information to them, which limits the functionality of the *GAS* model. For example, the S-V algorithm to be described in Section 5.3.1 is hard to be implemented in GraphLab.

GraphLab maintains a global scheduler, and workers fetch vertices from the scheduler for processing, possibly adding the neighbors of these vertices into the scheduler. The GraphLab engine executes the user-defined *GAS* function on each active vertex until no vertex remains in the scheduler. The GraphLab scheduler determines the order to activate vertices, which enables GraphLab to provide with both synchronous and asynchronous scheduling.

Asynchronous execution. Unlike the behaviors in a synchronous model, changes made to each vertex and edge during the *Apply* phase are committed immediately and visible to subsequent computation. Asynchronous execution can accelerate the convergence of some algorithms. For example, the *PageRank* algorithm can converge much faster with asynchronous execution. However, asynchronous execution may incur extra cost due to locking/unlocking and intertwined computation/communication.

Synchronous execution. GraphLab also provides a synchronous scheduler, which executes the *GAS* phases in order as an iteration. The *GAS* function of each active vertex runs synchronously with a barrier at the end of each iteration. Changes made to the vertex value is committed at the end of each iteration. Vertices activated in each iteration are executed in the subsequent iteration.

Vertex-cut partitioning. PowerGraph partitions an input graph by cutting the vertex set, so that the edges of a high-degree vertex are handled by multiple workers. As a tradeoff, vertices are replicated across workers, and communication among workers are required to guarantee that the vertex value on each replica remains consistent.

5.2.6 GraphChi

GraphChi [27] is implemented in C/C++, which is a single-machine system that can process massive graphs from secondary storage. In addition to the vertex-centric model, GraphChi introduces two new techniques to process large graphs in a single PC.

Out-of-core computation. An innovative out-of-core data structure is used to reduce the amount of random access to secondary storage. The parallel sliding windows algorithm partitions the input graph into subgraphs, called shards. In each shard, edges are sorted by the source IDs and loaded into memory sequentially.

Selective scheduling. GraphChi supports selective scheduling in order to converge faster on some parts of the graph, particularly when the change on values is significant. Each vertex in the *update*() function (similar to *apply*() in GraphLab) can add its neighbors to the scheduler and conduct selective computation.

5.2.7 Other Systems

In this chapter, we conduct experimental evaluation on Giraph [1], GraphLab/PowerGraph [30, 16], GPS [40], and Pregel+ [55], which we have discussed in Sections 5.2.2-5.2.5 and we also give an overview of various features supported by these systems in Figure 1.4. There are also a number of other systems that we do not evaluate experimentally, which we explain in this subsection.

Mizan [25] is a C++ optimized Pregel system that supports dynamic load balancing and vertex migration, based on runtime monitoring of vertices to optimize the end-to-end computation. However, Mizan performs pre-partitioning separately which takes much longer compared with Giraph and GPS, and the overhead of pre-partitioning can exceed the benefits. For this reason, we could not run Mizan on some large graphs used in this chapter and hence we do not include it in our experimental evaluation.

GraphX [53] is a graph parallel system, and it supports GraphLab and Pregel abstractions. Since GraphX is built upon the more general data parallel Spark system [57], the end-to-end performance of pipelined jobs can be superior when they are all implemented in Spark. Consider the task of first extracting the link graph from Wikipedia, and then computing PageRank on the link graph. Compared with implementing and running both jobs in Spark (with the second job done by GraphX), the traditional method of running the first (second) job by Hadoop (by a vertex-centric system) is more costly since it requires that the output of the first job be dumped to HDFS and reloaded by the second job. However, if only graph computation time is considered, GraphX is generally slower than GraphLab as reported in [53].

5.3 Algorithms

We use seven graph algorithms, which are classified into five categories based on the behaviors of the *compute* function in Pregel-like systems and the *GAS* function in Graphlab and PowerGraph.

Always-active. An algorithm is always-active if every vertex in every superstep sends messages to all its neighbors. Thus, the distribution of messages sent/received by all active vertices is the same across supersteps, i.e., the communication workload of every machine remains the same. Typical examples include *PageRank* [33] in synchronous computation and *Diameter Estimation* [21].

GraphLab's async algorithms. This category is specifically for algorithms that are designed to run on GraphLab's (also PowerGraph's) asynchronous computation model. Such algorithms add vertices to the scheduler, and then workers fetch vertices from the scheduler and pull data from neighboring edges and vertices for processing. The asynchronous execution can accelerate the convergence of algorithms such as the asynchronous *PageRank* and *Graph Coloring* [41] algorithms for GraphLab and PowerGraph.

Graph traversal. Graph traversal is a category of graph algorithms for which there is a set of starting vertices, and other vertices are involved in the computation based on whether they receive messages from their in-neighbors. Attribute values of vertices are updated and messages are propagated along the edges as the algorithm traverses the graph. Algorithms such as *HashMin* [37] and *Single-Source Shortest Paths* [33] are in this category.

Multi-phase. For this category of algorithms, the entire computation can be divided into a number of phases, and each phase consists of some supersteps. For example, in *Bipartite Maximal Matching* [2], there are four supersteps to simulate a three-way handshake in each phase. The *SV* [44] algorithm also falls into this category, since it simulates tree hooking and star hooking in each phase.

Graph mutation. Algorithms in this category need to change the topological structure of the input graph through edges and/or vertices addition and/or deletion. For example, the greedy graph coloring algorithm that removes a maximal independent set from the current graph iteratively falls into this category. Systems that do not support edge deletion, such as GraphLab and PowerGraph, cannot straightforwardly support these algorithms.

5.3.1 Algorithm Description

We now describe the seven graph algorithms.

PageRank

Given a directed web graph $G = (V, E)$, where each vertex (page) v links to a list of pages $\Gamma_{out}(v)$, the problem is to compute the PageRank value, $pr(v)$, of each $v \in V$.

The typical PageRank algorithm [33] for Pregel-like systems works as follows. Each vertex v keeps two fields: $pr(v)$ and $\Gamma_{out}(v)$. In superstep 0, each vertex v initializes $pr(v) = 1$ and sends each out-neighbor of v a message with a value of $pr(v)/|\Gamma_{out}(v)|$. In superstep i ($i > 0$), each vertex v sums up the received PageRank values, denoted by sum, and computes $pr(v) = 0.15 + 0.85 \times sum$. It then distributes $pr(v)/|\Gamma_{out}(v)|$ to each of its out-neighbors. This process terminates after a fixed number of supersteps or the PageRank distribution converges.

The asynchronous version of PageRank for GraphLab and PowerGraph, named as *Async-PageRank*, works as follows. Each vertex v keeps three fields: $pr(v)$, $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, where $pr(v)$ is initialized as 1. We define the generalized sum in the *GAS* function as a numerical sum. Then for each active vertex v fetched from the scheduler, in the *Gather* phase, the values $pr(u)/|\Gamma_{out}(u)|$ for all neighbors $u \in \Gamma_{in}(v)$ are gathered and summed up as sum. In the *Apply* phase, we update $pr(v) = 0.15 + 0.85 \times sum$. In the *Scatter* phase, if the change in value of $pr(v)$ is greater than ϵ (typically, ϵ is set to 0.01), we add each vertex $u \in \Gamma_{out}(v)$ to the scheduler. This process terminates after a fixed number of supersteps.

Diameter Estimation

Given a graph $G = (V, E)$, we denote the distance between u and v in G by $d(u, v)$. We define the neighborhood function $N(h)$ for $h = 0, 1, \ldots, \infty$ as the number of pairs of vertices that can reach each other in h hops or less.

Each vertex v keeps two fields: $N[h; v]$ and $\Gamma_{out}(v)$, where $N[h; v]$ indicates the set of vertices v can reach in h hops. In superstep 0, each vertex v sets $N[0; v]$ to $\{v\}$, and broadcasts $N[0; v]$ to each $u \in \Gamma(v)$. In superstep i ($i > 0$), each vertex v receives messages from its neighbors and set the value of $N[i; v]$ as the union of $N[i - 1; v]$ and $N[i - 1; u]$ for all $u \in \Gamma(v)$. A global aggregator is used to compute the total pairs of vertices, denoted by $N(i)$, that can be reached from each other after superstep i. The algorithm terminates if the following stop condition is true: in superstep i, $N(i)$ is less than or equal to $(1 + \epsilon) * N(i - 1)$.

To handle the large volume of each vertex's neighborhood information, i.e., $N[h; v]$, the algorithm applies the idea of Flajolet-Martin [14], which was also used in the ANF algorithm [35].

Single-Source Shortest Paths (SSSP)

Let $G{=}(V, E)$ be a weighted graph, where each edge $(u, v){\in}E$ has length $\ell(u, v)$. The length of a path P is equal to the sum of the length of all the edges on P. Given a source $s \in V$, the SSSP algorithm computes a shortest path from s to every other vertex $v \in V$, denoted by $SP(s, v)$, as follows. Each vertex v keeps two fields: $\langle prev(v), dist(v)\rangle$ and $\Gamma_{out}(v)$, where $prev(v)$ is the vertex preceding v on $SP(s, v)$ and $dist(v)$ is the length of $SP(s, v)$. Each out-neighbor $u \in \Gamma_{out}(v)$ is also associated with $\ell(v, u)$.

Initially, only s is active with $dist(s) = 0$, and $dist(v) = \infty$ for any other vertex v. In superstep 0, s sends a message $\langle s, dist(s) + \ell(s, u)\rangle$ to each $u \in \Gamma_{out}(s)$, and votes to halt. In superstep i ($i > 0$), if a vertex v receives messages $\langle w, d(w)\rangle$ from any of v's in-neighbor w, then v finds the in-neighbor w^* such that $d(w^*)$ is the smallest among all $d(w)$ received. If $d(w^*) < dist(v)$, v updates $\langle prev(v), dist(v)\rangle = \langle w^*, d(w^*)\rangle$, and sends a message $\langle v, dist(v) + \ell(v, u)\rangle$ to each out-neighbor $u \in \Gamma_{out}(v)$. Finally, v votes to halt.

HashMin

Assume each CC C in an undirected graph G has a unique ID, and for each vertex v in C, let $cc(v)$ be the ID of C. Given G, HashMin [37] computes $cc(v)$ for each v in G, and hence all CCs for G, as all vertices with the same $cc(v)$ form a CC.

Each vertex v keeps two fields: $min(v)$ and $\Gamma(v)$, where $min(v)$ is initialized as the ID of the vertex itself. HashMin broadcasts the smallest vertex ID seen so far by each vertex v as follows. In superstep 0, each vertex v sets $min(v)$ to be the smallest ID among $id(v)$ and $id(u)$ of all $u \in \Gamma(v)$, broadcasts $min(v)$ to all its neighbors, and votes to halt. In superstep i ($i > 0$), each vertex v receives messages from its neighbors; let min^* be the smallest ID received, if $min^* < min(v)$, v sets $min(v) = min^*$ and broadcasts min^* to its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, $min(v) = cc(v)$ for all v.

Shiloach-Vishkin's Algorithm (SV)

The HashMin algorithm requires $O(\delta)$ supersteps for computing CCs, which is too slow for graphs with a large diameter, such as spatial networks where $\delta \approx O(\sqrt{n})$. The SV algorithm [44] can be translated into a Pregel algorithm which requires $O(\log n)$ supersteps [56], which can be much more efficient than HashMin for computing CCs in general graphs.

The SV algorithm groups vertices into a forest of trees, so that all vertices in each tree belong to the same CC. The tree here is relaxed to allow the root to have a self-loop. Each vertex v keeps two fields: $D[u]$ and $\Gamma_{out}(u)$, where $D[u]$ points to the parent of u in the tree and is initialized as u (i.e., forming a self loop at u).

The SV algorithm proceeds in phases, and in each phase, the pointers are updated in the following three steps: (1)for each edge (u, v), if u's parent w is the root, set w as a child of $D[v]$, which merges the tree rooted at w into v's tree; (2)for each edge (u, v), if u is in a star, set u's parent as a child of $D[v]$; (3)for each vertex v, set $D[v] = D[D[v]]$. We perform Steps (1) and (2) only if $D[v] < D[u]$, so that if u's tree is merged into v's tree due to edge (u, v), then edge (v, u) will not cause v's tree to be merged into u's tree again. The algorithm ends when every vertex is in a star.

Bipartite Maximal Matching (BMM)

Given a bipartite graph $G = (V, E)$, this algorithm computes a BMM, i.e., a matching to which no additional edge can be added without sharing an end vertex. The algorithm [33] proceeds in phases, and in each phase, a three-way handshake is simulated.

Each vertex v keeps three fields: $S[v]$, $M[v]$ and $\Gamma_{out}(v)$, where $S[v]$ indicates which set the vertex is in (L or R) and $M[v]$ is the name of its matched vertex (initialized as -1 to indicate that v is not yet matched).

The algorithm computes a three-way handshake in four supersteps: (1)each vertex v, where $S[v] = L$ and $M[v] = -1$, sends a message to each of its neighbors $u \in \Gamma_{out}(v)$ to request a match; (2)each vertex v, where $S[v] = R$ and $M[v] = -1$, randomly chooses one of the messages w it receives, sends a message to w granting its request for match, and sends messages to other requestors $w' \neq w$ denying its request; (3)each vertex v, where $S[v] = L$ and $M[v] = -1$, chooses one of the grantors w it receives and sends an acceptance message to w; (4)each vertex v, where $S[v] = R$ and $M[v] = -1$, receives at most one acceptance message, and then changes $M[v]$ to the acceptance message's value. All vertices vote to halt at the end of a superstep.

Graph Coloring (GC)

Given an undirected graph $G = (V, E)$, GC computes a color for every vertex $v \in V$, denoted by $color(v)$, such that if $(u, v) \in E$, $color(u) \neq color(v)$.

The GC algorithm for Pregel-like systems normally adopts the greedy GC algorithm from [16]. The algorithm iteratively finds a maximal independent set (MIS) from the set of active vertices, assigns the vertices in the MIS a new color, and then removes them from the graph, until no vertices are left in the graph. Each iterative phase is processed as follows, where all vertices in the same MIS are assigned the same color c: (1)each vertex $v \in V$ is selected as a tentative vertex in the MIS with a probability $1/(2 * |\Gamma(v)|)$; if a vertex has no neighbor (i.e. an isolated vertex or becoming isolated after graph mutation), it is a trivial MIS; each tentative vertex v then broadcasts $id(v)$ to all its neighbors; (2)each tentative vertex v receives messages from its tentative neighbors; let min^* be the smallest ID received, if $min^* > id(v)$, then v is included in the MIS and $color(v) = c$, and $id(v)$ is broadcast to its neighbors; (3)if a vertex u receives messages from its neighbors (that have been included in the MIS in superstep (2)), then for each such neighbor v, delete v from $\Gamma(u)$.

5.3.2 Algorithmic Optimizations

Apart from algorithm categorization, we also describe three algorithmic optimizations [41] here that can improve the performance of distributed graph computing systems on certain algorithms.

Finishing Computations Serially (FCS). Some algorithms may run for a large number of supersteps, even though the later supersteps are merely executing on a small fraction of the graph, called the *active-subgraph*. FCS monitors the size of the active-subgraph and sends it to the master for serial computation as soon as the size is below a threshold (5M edges by default [41]), so as to terminate the computation earlier without running a prolonged number of supersteps. The results computed in the master are then sent back to the workers. FCS can be applied to algorithms in which an inactive vertex will not be activated again in later process. Among the algorithms we discussed, BMM and GC have this property.

Edge Cleaning On Demand (ECOD). Edge cleaning in an algorithm removes edges from the graph. ECOD delays the operation of edge cleaning and regards the edges as *stale edges* until they are involved in later computation where they are demanded to be removed.

Single Pivot (SP). SP is a heuristic for speeding up the computation of connected components (CC). SP first samples a vertex v called the *pivot*, and runs the cheaper BFS algorithm instead of the CC algorithm from v. Then, a standard CC algorithm is run on the graph excluding the CC that contains v. Theoretically, v has a higher probability to be in the giant CC of the graph, and the graph excluding the giant CC can be much smaller.

	Data	Type	|V|	|E|	AVG Deg	Max Deg
Web graphs	**WebUK**	directed	133,633,040	5,507,679,822	41.21	22,429
Social networks	**Friendster**	undirected	65,608,366	3,612,134,270	55.06	5,214
	Twitter	directed	52,579,682	1,963,263,821	37.33	779958
	LiveJournal	undirected	10,690,276	224,614,770	21.01	1,053,676
RDF	**BTC**	undirected	164,732,473	772,822,094	4.69	1,637,619
Spatial networks	**USA Road**	undirected	23,947,347	58,333,344	2.44	9

Figure 5.1: Datasets

5.4 Experimental Evaluation

We now evaluate the performance of *Giraph*, *GPS*, *Pregel+*, *GraphLab* (we use GraphLab 2.2 which includes all the features of PowerGraph), and use *GraphChi* as a single machine baseline. We release all the source codes of the algorithms used in our evaluation in `www.cse.cuhk.edu.hk/pregelplus/exp`, while the source codes of the different systems can be found in their own websites.

Datasets. We used six large real-world datasets, which are from four different domains as shown in Figure 5.1: (1)web graph: WebUK[1]; (2)social networks: Friendster[2], LiveJournal (LJ)[3] and Twitter[4]; (3)RDF graph: BTC[5]; (4)road networks: USA[6]. Among them, WebUK, LJ, Twitter and BTC have skewed degree distribution; WebUK, Friendster and Twitter have average degree relatively higher than other large real-world graphs; USA and WebUK have a large diameter, while Friendster, Twitter and BTC have a small diameter.

We also used synthetic datasets for scalability tests, where we generate power law graphs using Recursive Matrix (R-MAT) model [3] and random graphs using PreZER algorithm [34].

Experimental settings. We ran our experiments on a cluster of 15 machines, each with 48 GB DDR3-1,333 RAM, two 2.0GHz Intel(R) Xeon(R) E5-2620 CPU, a SATA disk(6Gb/s, 10k rpm, 64MB cache) and a Broadcom Gigabit Ethernet BCM5720 network adapter, running 64-bit CentOS 6.5 with Linux kernel 2.6.32. Giraph 1.0.0 and GPS (rev. 112) are built on JDK 1.7.0 Update 45, the hadoop DFS is built on Apache Hadoop 1.2.1. Pregel+, GraphChi (2014.4.30) and GraphLab 2.2 are compiled using GCC 4.4.7 with -O2 option enabled, and MPICH 3.0.4 is used.

[1] `http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05`

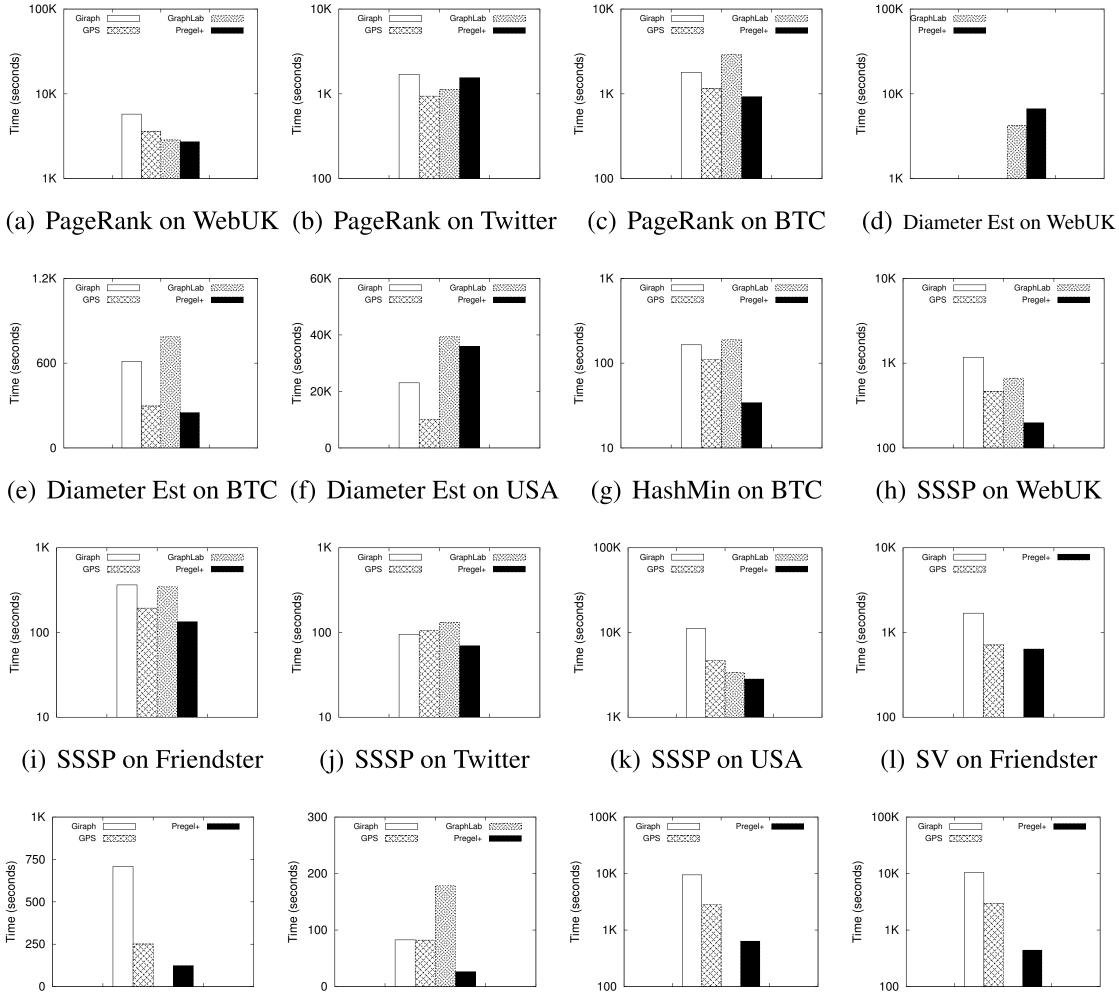
[2] `http://snap.stanford.edu/data/com-Friendster.html`

[3] `http://konect.uni-koblenz.de/networks/livejournal-groupmemberships`

[4] `http://konect.uni-koblenz.de/networks/twitter_mpi`

[5] `http://km.aifb.kit.edu/projects/btc-2009/`

[6] `http://www.dis.uniroma1.it/challenge9/download.shtml`



(a) PageRank on WebUK (b) PageRank on Twitter (c) PageRank on BTC (d) Diameter Est on WebUK

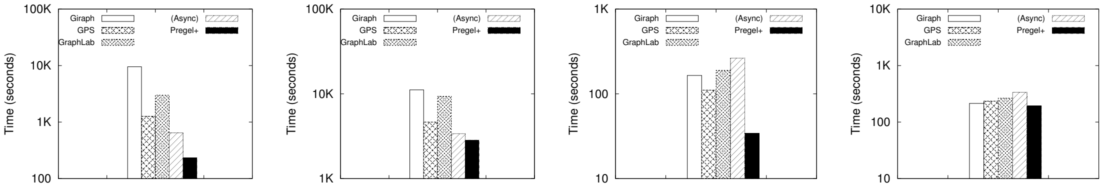
(e) Diameter Est on BTC (f) Diameter Est on USA (g) HashMin on BTC (h) SSSP on WebUK

(i) SSSP on Friendster (j) SSSP on Twitter (k) SSSP on USA (l) SV on Friendster

(m) SV on USA (n) BMM on LJ (o) Coloring on Friendster (p) Coloring on BTC

Figure 5.2: Performance overview on Giraph, GPS, GraphLab, and Pregel+

Unless otherwise stated, we use all 15 machines for all distributed systems for all experiments; and we use 8 cores in each machine for GraphChi, Giraph and GraphLab which run with multithreading, and 8 processes (also using 8 cores) in each machine for GPS and Pregel+. There is no limit set on the amount of memory each system can use, i.e., all the systems have access to all the available memory (48GB) in each machine. GPS's polling time is set to 10 ms in order to reduce the fixed overhead in each superstep (more details can be found in our technical report [32]), while the threshold for LALP is set to 100. The fault tolerance mechanisms of all the systems are off by default. All other



(a) HashMin on USA (b) SSSP on USA (c) HashMin on BTC (d) HashMin on Friendster

Figure 5.3: Performance of asynchronous computing (in GraphLab) and synchronous computing

settings, if any, of the systems are as their default. All running time reported is the wall-clock time elapsed during loading and computing, but not including dumping since this is identical for all the distributed systems.

Objectives. We evaluate the systems following the key objectives listed in Section 1.4, and provide detailed comparative analysis on each of the evaluation criteria.

5.4.1 System Performance Overview

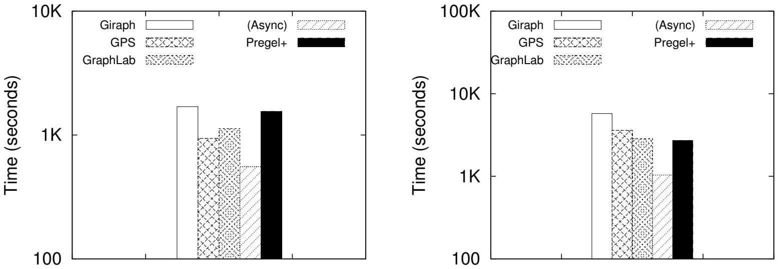
We first evaluate (1)the performance of the various systems w.r.t. different algorithm categories (discussed in Section 5.3), (2)the performance of the various systems on graphs with different characteristics, and (3)the performance of a distributed system compared with that of a baseline single-machine system. The results to be presented in Subsections 5.4.1–5.4.1 give readers an overview on the performance of the various systems.

In the experiments in Sections 5.4.1–5.4.1, we enable the techniques of the various systems that give the best performance, while in Sections 5.4.2–5.4.5 we analyze the effects of each individual technique. We run the systems on every algorithm-graph combination that makes sense (except for HashMin, SV, and Coloring which are applicable on undirected graphs only, and Bipartite Maximal Matching which runs on bipartite graphs only). We also do not report all system-algorithm combinations, since it is not clear how pointer jumping in SV and edge deletion in graph coloring can be implemented in GraphLab.

With limited space, we only report 16 figures (in Figures 5.2(a)–5.2(p)) that are sufficient to reflect the overall performance over all the figures (reported in Figures 3–9 in our technical report [32]).

Performance on Different Algorithms

We first analyze the performance of the various systems w.r.t. different algorithm categories.



(a) PageRank on Twitter (b) PageRank on WebUK

Figure 5.4: Asynchronous PageRank in GraphLab

Performance on always-active algorithms. Figures 5.2(a)–5.2(f) report the performance of the systems on two representative always-active algorithms, i.e., synchronous PageRank and Diameter Estimation. It is difficult to draw a clear conclusion on which system has the best performance. Overall, GPS and Pregel+ have better performance in most cases. Between Giraph and GraphLab, GraphLab has the best performance in some cases while Giraph ran out of memory for diameter estimation on WebUK. Moreover, taking into account all the results in Figures 3 and 4 in [32], GraphLab is faster than Giraph in more cases.

Performance on graph-traversal algorithms. Figures 5.2(g)–5.2(k) report the performance of the systems on two representative graph-traversal algorithms, i.e., HashMin and SSSP. The results show that Pregel+ clearly outperforms all the other systems (Figures 5 and 6 in [32] show the same trend). GPS has better performance than Giraph and GraphLab in most cases.

Performance on multi-phase algorithms. Figures 5.2(l)–5.2(n) report the performance of the systems on two representative multi-phase algorithms, i.e., BMM and SV. Pregel+ always has the best performance for this category of algorithms, while GPS is faster than Giraph and GraphLab.

Performance on graph mutation. To test the performance on graph mutation, we use the graph coloring algorithm. We do not report GraphLab since it does not support edge deletion. Figures 5.2(o) and 5.2(p) (also Figure 9 in [32]) show that Pregel+ is much faster than both GPS and Giraph for graph coloring.

Performance on GraphLab's async mode. GraphLab supports both synchronous and asynchronous execution and we evaluate the performance as follows. First, for a graph with a large diameter, e.g., USA road network, Figures 5.3(a)–5.3(b) show that asynchronous execution is significantly faster than synchronous execution. This is because changes made to each vertex and edge during the *apply* phase in asynchronous mode

		Overall performance ranking (1: best)		
		1	**2**	**3**
Always active		Pregel+/GPS	GraphLab	Giraph
Graph traversal		Pregel+	GPS	GraphLab/Giraph
Multi-phase	SV	Pregel+	GPS	Giraph
	BMM	Pregel+	GPS/Giraph	GraphLab
Graph mutation		Pregel+	GPS	Giraph

Figure 5.5: Overall performance on different algorithms

are committed immediately and visible to subsequent computation; while in synchronous mode, the change commits are delayed till the end of each superstep, leading to slower convergence. However, for processing graphs with a small diameter as shown in Figures 5.3(c)–5.3(d), the overhead of locking/unlocking is not paid off by the faster convergence of asynchronous execution.

However, for some algorithms, asynchronous execution can lead to faster convergence even for small-diameter graphs. For example, for asynchronous PageRank, most vertices can converge after only a small number of updates. On the contrary, in synchronous execution, all vertices need to update their PageRank values and distribute their new values to neighbors. In each superstep, there are $O(n)$ updates made and $O(m)$ messages transmitted. In asynchronous execution, the global scheduler only maintains the vertices that need to be updated. If there is a significant change in some vertex's PageRank value, then it activates its neighbors and puts them into the global scheduler. Figures 5.4(a)–5.4(b) show that PageRank takes only 554.4 seconds on the small-diameter Twitter graph using 508,251,513 updates, and takes 1,037.9 seconds on the large-diameter WebUK graph using 847,312,369 updates. However, the synchronous PageRank uses 4,679,591,698 and 11,893,340,560 updates, respectively, and are also much slower.

Overall performance. Figure 5.5 gives a ranking on the overall performance of the systems for different algorithm categories. Pregel+ and GPS have superior performance for always-active algorithms, because those algorithms generate a lot of messages, which is effectively addressed by Pregel+ and GPS's message reduction techniques (see Section 5.4.3). Graph-traversal algorithms also generate a large number of messages from active vertices (especially high-degree ones), usually in the preceding supersteps. Pregel+ has better performance than GPS because of its integration of mirroring and combiner, which is more effective than LALP alone in GPS. Although vertex replica in GraphLab is similar to mirroring, such replica is constructed for every vertex instead of just high-degree vertices and so the extra overhead is not paid off. For SV, Pregel+'s request-response technique is effective (see Section 5.4.4). But in general, no system has any specific technique to improve its performance on multi-phase algorithms, though the message reduction techniques still help improve the performance. For graph coloring, GPS and Giraph require users to subclass a separate Edge Class during the graph loading phase and edge deletion requests must be made in the *compute*() function; while in Pregel+, the edge information of a graph is stored with vertices, which simplifies the API and enables faster edge addition/deletion.

C++ v.s. Java. Among the systems, Giraph and GPS are implemented in Java, and GraphLab and Pregel+ are implemented in C++. While it is difficult to tell from our results which language, C++ or Java, leads to better performance, Figure 5.2(d) shows that the Java-based systems ran out of memory on the large WebUK graph for diameter estimation (GPS also ran out of memory on Friendster as shown in Figure 4(b) in [32]). This is mainly because a Java object takes more space than a C++ object; moreover, Java uses Garbage Collector to automatically handle object deletion, which cannot keep the pool of objects small in an optimal manner and hence often leads to larger memory usage. As for running time, none of the distributed systems has an implementation in both Java and C++, and hence we cannot make a comparison that gives a clear conclusion. But Java-based systems incur extra (de)serialization cost for processing objects in binary representation in memory. Moreover, the single-machine system GraphChi was implemented in both Java and C++, and [27] remarks that the Java implementation ran 2-3 times slower than the C++ implementation. Thus, we believe Pregel+'s C++ implementation also contributes to its superior overall performance.

Performance on Different Graphs

Next we analyze the performance of the various systems on graphs with different characteristics.

Performance on graphs with skewed degree distribution. Figures 5.2(a), 5.2(b), 5.2(c), 5.2(d), 5.2(e), 5.2(g), 5.2(h), 5.2(j), 5.2(n) and 5.2(p) report the performance of the systems on graphs with skewed degree distribution (i.e., WebUK, LJ, BTC, and Twitter). The results show that Pregel+ (thanks to its mirroring and request-respond techniques) has the best performance in most cases, while GPS (with the help of LALP) also has good performance in most of the cases. GraphLab is faster than Giraph in about half of the cases but is slower in the rest. Overall, no system is always better than the others in processing graphs with skewed degree distribution, but Pregel+ and GPS are the better choices as they exhibit good performance in most of the cases tested.

Performance on graphs with a large diameter. Figures 5.2(a), 5.2(d), 5.2(f), 5.2(h), 5.2(k) and 5.2(m) report the performance of the systems on graphs with a large diameter (e.g., WebUK and USA Road). Again, each system has better performance in some cases, but overall Pregel+ has the best performance in more cases. GPS and GraphLab beat each other in roughly equal number of cases, while Giraph has the worst performance in most cases.

Performance on graphs with a small diameter. Figures 5.2(b), 5.2(c), 5.2(e), 5.2(g), 5.2(i), 5.2(j), 5.2(l), 5.2(o) and 5.2(p) report the performance of the systems on graphs with a small diameter, e.g., Friendster, Twitter and BTC. Pregel+ has the best performance in most cases, while GPS also has good performance in most cases. Giraph has poorer performance than GPS, but is better than GraphLab overall.

Performance on graphs with high average degree. Figures 5.2(a), 5.2(b), 5.2(d), 5.2(h), 5.2(i), 5.2(j), 5.2(l) and 5.2(o) report the performance of the systems on graphs with a relatively high average degree (e.g., WebUK, Friendster and Twitter). Pregel+ has the best performance in most cases. GPS is generally faster than GraphLab, while both of them are faster than Giraph in most cases.

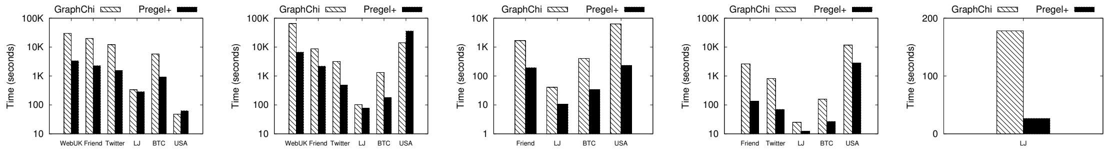
Overall performance. Figure 5.6 gives a ranking on the overall performance of the systems for different types of graphs. Pregel+ has the best performance in most cases mainly because of its integration of mirroring and combiner, which effectively addresses load balancing in skewed-degree graphs and reduces messages in graphs with high average degree. Similarly, GPS's LALP is also effective in addressing load balancing and in message reduction. GraphLab's vertex-cut partitioning effectively addresses load balancing, but it has an overhead of locking/unlocking (which is required even in synchronous mode, e.g., to prevent more than one vertices from scattering values to the same vertex), and hence its overall performance is not much better than Giraph. For handling large-diameter graphs which usually require a large number of supersteps, our results reveal that systems like Giraph, which have a large constant overhead (hundreds of ms) per superstep, can be very slow. On the contrary, Pregel+ has a very small constant overhead per superstep, while GPS and GraphLab have a slightly larger constant overhead per superstep than that of Pregel+. Finally, small diameter usually leads to faster convergence and hence for systems like GraphLab, which has a larger start-up overhead in vertex-cut partitioning, can be slower than other systems.

Comparison with GraphChi

We also compare with a single-machine baseline, GraphChi [27]. Due to space limitation, we only report the performance of Pregel+ and GraphChi in Figures 5.7(a)–5.7(e). Performance of other systems can be compared by referring to the performance of Pregel+ and other systems in Figures 5.2(a)–5.2(p) (and Figures 3–9 in [32]). GraphChi needs to

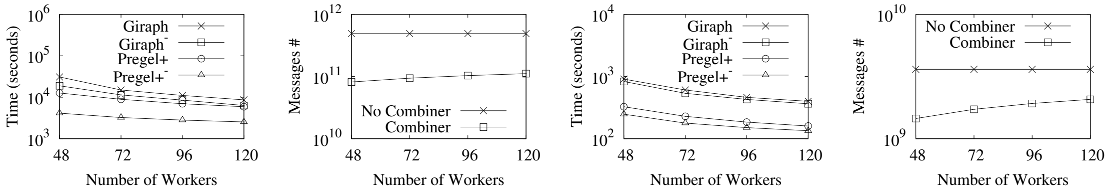
	Overall performance ranking (1: best)			
	1	**2**	**3**	**4**
Skewed degree	Pregel+	GPS	GraphLab / Giraph	-
Large diameter	Pregel+	GPS / GraphLab	Giraph	-
Small diameter	Pregel+	GPS	Giraph	GraphLab
High average degree	Pregel+	GPS	GraphLab	Giraph

Figure 5.6: Overall performance on different graphs



(a) PageRank (b) Diameter Est. (c) HashMin (d) SSSP (e) BMM

Figure 5.7: Performance of GraphChi v.s. Pregel+ on PageRank, Diameter Est., HashMin, SSSP and BMM



(a) PageRank on WebUK (b) PageRank on WebUK (c) SSSP on Friendster (d) SSSP on Friendster

Figure 5.8: Effects of combiner in Giraph and Pregel+ with different number of workers

pre-sort the graph, the cost of which is reported in Figure 16 in [32].

We do not run SV and GC since it is not clear to us how pointer jumping in SV and edge deletion in GC can be implemented in GraphChi. We also note that GraphChi took much longer to run SSSP on WebUK and we killed the job after its running time is three orders of magnitude longer than Pregel+'s.

The results show that Pregel+ is about 10 times faster than GraphChi when processing the large graphs, WebUK, Friendster, Twitter and BTC. But for the two smaller graphs, LJ and USA, which can fit in the memory of a single machine, GraphChi uses fully main-memory mode and its running time is closer to that of Pregel+ (and even faster in two cases on USA). Thus, GraphChi is a reasonable choice for moderate-sized graphs. But when the graph is large and sufficient computing resources are available, a distributed system can achieve much better performance than a single-machine system. To be more specific, Pregel+ requires 4, 3, 2 and 2 machines to process WebUK, Friendster, Twitter and BTC in memory, respectively; and given such number of machines, Pregel+ is already much faster than GraphChi (see details in Section 5.4.1 in [32]). We note that Java-based systems such as Giraph and GPS may require more machines as they use more memory.

5.4.2 Effects of Message Combiner

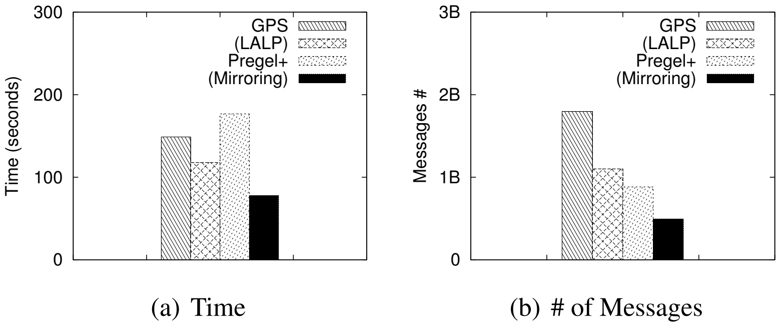
We now study the effects of message combiner. GPS does not perform sender-side message combining, as the authors claim that very small performance difference can be observed whether combiner is used or not [40]. To verify whether this claim is valid, we first analyze how many messages can be combined by applying a message combiner as follows (the proof is detailed in [32]).

Theorem 5.4.1. *Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we assume that the vertex set is evenly partitioned among M machines (i.e., each machine holds n/M vertices). We further assume that the neighbors of a vertex in G are randomly chosen among V, and the average degree $deg_{avg} = m/n$ is a constant. Then, at least $(1 - \exp\{-deg_{avg}/M\})$ fraction of messages can be combined using a combiner in expectation.*

According to Theorem 5.4.1, if a large number of machines are available and the average degree is small, then indeed applying combiner may not improve the performance much as claimed in [40]. For example, if $M = 1000$ and $deg_{avg} = 10$, then only 1% of the messages can be combined. However, in many applications and for many datasets (e.g., for all the algorithms we discuss in this chapter and graphs with more than 5 billions of edges we used here), one may not require or use thousands of machines. When M is smaller, combiners can effectively reduce the number of messages to be sent over the network and hence improve the performance of the systems, which we verify as follows.

We assess the effect of combiner by testing the two systems, Giraph and Pregel+, that support combiner. We use two versions for each system, Giraph vs Giraph^{-} and Pregel+ vs Pregel+$^{-}$, where the superscript '$^{-}$' indicates that combiner is not applied. As shown in Figures 5.8(b) and 5.8(d), there is an obvious reduction on the total number of messages sent over the network when combiner is applied. As the number of machines increases, less messages are combined but the number is still considerably smaller than that without combiner.

Figures 5.8(a) and 5.8(c) further show that the running time of both systems with combiner is shorter than that without combiner. In conclusion, applying combiner can always



(a) Time (b) # of Messages

Figure 5.9: Diameter Est on LJ (LALP and mirroring)

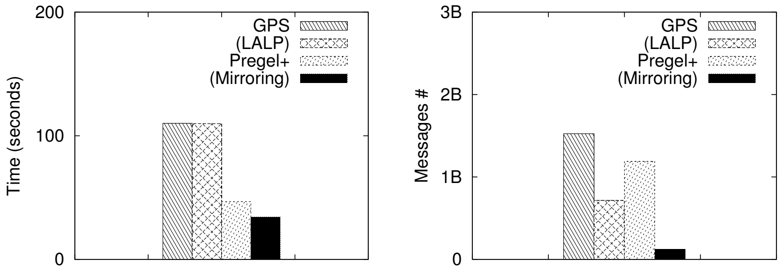
reduce the total number of messages and shorten the running time. Although the improvement is not so obvious in some cases, there also exist cases where the improvement is quite significant, e.g., running PageRank in Pregel+$^-$ on WebUK. This conclusion has also been verified on many other algorithms on the datasets we used, and we have not found a case where applying combiner leads to worse performance than without combiner.

5.4.3 Effects of LALP and Mirroring

We now study the effects of LALP in GPS and mirroring in Pregel+. We report the performance of GPS and Pregel+, with and without LALP/mirroring, for running Diameter Estimation on LJ in Figure 5.9 and for running HashMin on BTC in Figure 5.10. The results show that applying LALP/mirroring reduces the running time in both cases. For running Diameter Estimation on LJ, using LALP in GPS is 1.3 times faster and using mirroring in Pregel+ is 2.2 times faster than without using the techniques. For running HashMin on BTC, the reduction in running time is not as significant, because the number of supersteps that involve a large number of redundant messages is only 4, and so message reduction is significantly smaller than in Diameter Estimation where many supersteps involves a large number of redundant messages. Figures 22 and 23 in [32] further show the skewed distribution in the number of messages sent by the workers of GPS/Pregel+ is evened by applying LALP/mirroring. In addition, there is also a significant reduction in the number of messages sent by each worker.

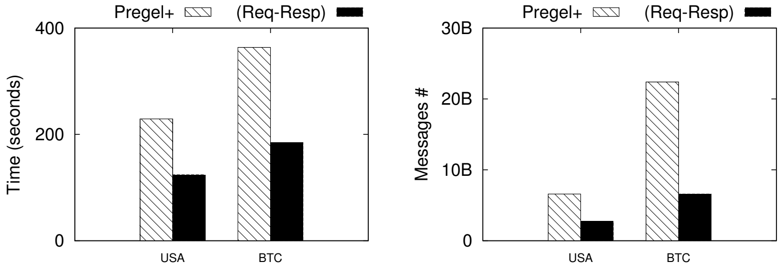
5.4.4 Effects of Request-Respond API

We next study the effects of the request-respond technique in Pregel+, which can address the imbalanced workload created by algorithm logic such as in the SV algorithm. We test



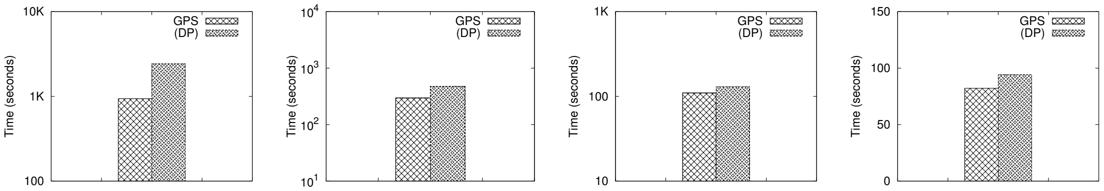
(a) Time (b) # of Messages

Figure 5.10: HashMin on BTC (LALP and mirroring)



(a) Time (b) # of Messages

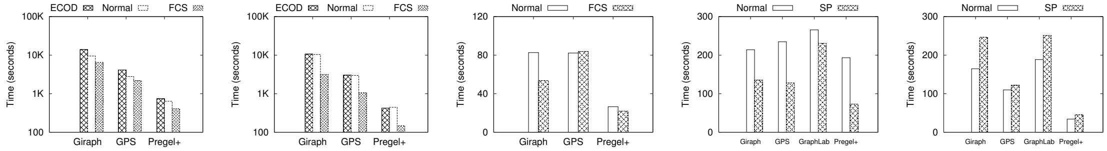
Figure 5.11: Effects of request-respond API in Pregel+



(a) PageRank on Twitter (b) Diameter Est. on BTC (c) HashMin on BTC (d) BMM on LJ

Figure 5.12: Effects of dynamic repartitioning

SV on the USA road network and the BTC graph. Figure 5.11(a) shows that the running time of Pregel+ is almost reduced by half after applying the request-respond technique, which can be explained by the significant reduction in the number of messages as shown in Figure 5.11(b). Figure 24 in [32] further shows that the imbalanced communication workload caused by the logic of SV is effectively eliminated by the request-respond technique.



(a) GC on Friendster (b) GC on BTC (c) BMM on LJ (d) CC on Friendster (e) CC on BTC

Figure 5.13: Effects of ECOD, FCS and SP

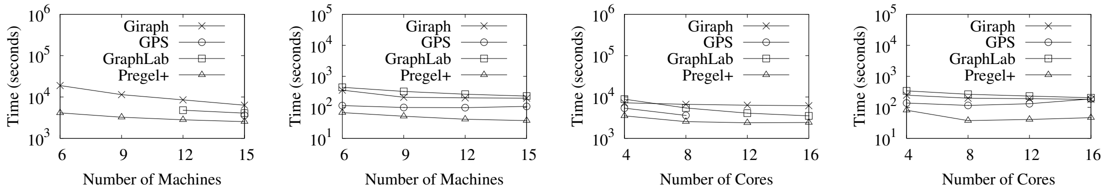
5.4.5 Effects of Dynamic Repartitioning

GPS also adopts a dynamic repartitioning (DP) technique to redistribute vertices across workers. The DP technique can be applied in all algorithms and on all graph types. We report the performances of PageRank on Twitter, Diameter Estimation on BTC, HashMin on BTC, and BMM on LiveJournal in Figures 5.12(a)–5.12(d). In all cases, DP does not obtain a good graph partition in the entire process, and therefore, the performances degrades due to the computational overhead incurred by the technique. As pointed out in [40], the benefit of DP can only be observed in very limited settings, e.g., very large number of supersteps of running PageRank computation. Therefore, it is difficult for DP to gain performance benefit in general, and we have tested many cases and have not found a case where DP can improve the performance.

5.4.6 Effects of Algorithmic Optimizations

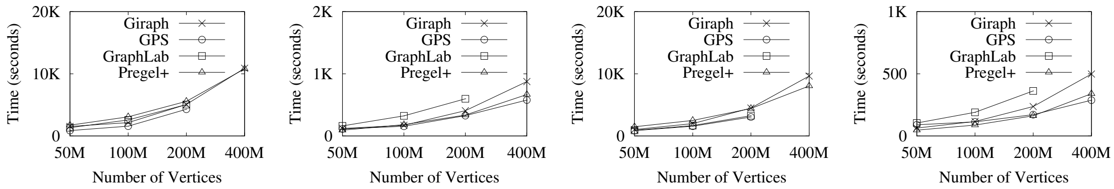
We next study the effects of the algorithmic optimizations described in Section 5.3.2. Figures 5.13(a)–5.13(c) report the effects of applying FCS and ECOD in Giraph, GPS and Pregel+ (more results can be found in Section 5.4.6 in [32]). Note that FCS and ECOD cannot be applied in GraphLab since it does not support edge deletion and Pregel-like aggregator. The results show that FCS considerably improves the performance of Graph Coloring (GC) in all the systems, since GC takes a large number of supersteps to converge on Friendster and BTC, and hence FCS can significantly reduce the number of supersteps. Figure 5.13(c) shows that FCS also improves the performance of the systems on BMM. On the other hand, the results show that ECOD degrades the performance. This is mainly because GC removes every stale edge in later computation and hence ECOD does not reduce the total workload. Moreover, ECOD incurs additional overhead. Thus, ECOD is only effective in algorithms in which many stale edges will not be touched again after they are decided to be removed [41].

Figures 5.13(d)–5.13(e) (more results can be found in Section 5.4.6 in [32]) report the effects of applying SP in different systems for computing connected components (CC),



(a) PR: Effect of Machine# (b) CC: Effect of Machine# (c) PR: Effect of Core # (d) CC: Effect of Core #

Figure 5.14: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of machines or CPU cores



(a) PR: Random Graphs (b) CC: Random Graphs (c) PR: Power-law Graphs (d) CC: Power-law Graphs

Figure 5.15: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of vertices

where we use HashMin to compute CCs in the remaining graph after BFS from the pivot. The performance of the systems on Friendster is significantly improved, but degrades on BTC. The reason is that, all vertices in Friendster constitutes a single giant CC, and so no matter which vertex is chosen as the pivot, the computation will terminate after running BFS from the pivot. In other words, less costly BFS is ran on the whole graph instead of HashMin. However, the largest component of BTC consists of only around 3% of all the vertices. Thus, SP can only label a small fraction of the graph and the overhead exceeds the gain obtained by SP.

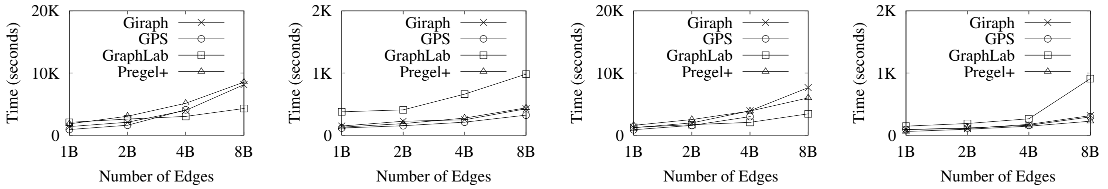
5.4.7 Scalability of Various Systems

We evaluate the scalability of the systems on both real-world graphs and synthetic graphs.

Effects of Number of Machines/CPU-Cores

We first report the performance of the systems by varying the number of machines or CPU cores.

Effects of machine number. We vary the number of machines from 6 to 15 in this experiment, and fix the number of CPU cores in each machine to 8. GPS and Pregel+ run



(a) PR: Random Graphs (b) CC: Random Graphs (c) PR: Power-law Graphs (d) CC: Power-law Graphs

Figure 5.16: Performance of Giraph, GPS, GraphLab, and Pregel+ with different number of edges

8 processes on each machine, while Giraph and GraphLab can take advantages of all the computing resource from the 8 cores by multithreading.

We first consider PageRank on WebUK, Figure 5.14(a) shows that only Giraph scales linearly with the number of machines, though it is significantly slower than the other systems. Pregel+ scales almost linearly (note that Figures 5.14(a)–5.14(d) are in logarithmic scale) and it is the fastest system in all settings. For GraphLab, we can only obtain the results when there are at least 12 machines in the cluster, as the total aggregate memory of the cluster is not sufficient for running GraphLab on this large web graph when there are less than 12 machines. The situation is similar for GPS, but for the smaller BTC graph, we obtain their results for all cases as reported in Figure 5.14(b). For processing the BTC graph, Giraph, GraphLab, and Pregel+ all scale linearly with the number of machines, but GPS's running time does not change much as the number of machines increases.

Effects of CPU core number. We vary the number of CPU cores in each machine from 4 to 16 in this experiment, and fix the number of machines in the cluster to 15. The number of processes in GPS and Pregel+ is the same as the number of CPU cores.

For running PageRank on WebUK, Figure 5.14(c) shows that only GraphLab scales sub-linearly with the number of CPU cores. This is because GraphLab can take advantage of multithreading. Giraph also uses multithreading but the effect is not obvious. The running time of GPS and Pregel+ decreases considerably (about 1.5 times) when the number of processes in each machine increases from 4 to 8, but further increasing the number of processes does not improve the performance since the overhead of network communication also increases with the number of processes.

For processing BTC, Figure 5.14(d) shows that multithreading in Giraph and GraphLab becomes even less effective since the dataset is much smaller. Pregel+ scales only linearly when the number of processes in each machine doubles from 4 to 8, while the performance of GPS even degrades when the number of processes in each machine increases.

Effects of Graph Size

We now report the performance of the systems by varying the number of vertices and edges using synthetically generated random graphs [34] and power-law graphs [3]. We set the number of machines in the cluster to be 15 and the number of CPU cores or processes in each machine to be 8.

Effects of vertex number. We vary the number of vertices in the synthetic graphs from 50M to 400M, while we fix the average vertex degree of each graph to 20. As Figures 5.15(a)–5.15(d) show, the running time of all the systems increases approximately linearly with the number of vertices. However, GPS ran out of memory when running PageRank on the two largest graphs, while GraphLab ran out of memory when running both PageRank and HashMin on the two largest graphs. Giraph and Pregel+ can run on all graphs, but Giraph has poorer scalability than Pregel+ when the graph size becomes larger.

Effects of edge number. We fix the number of vertices in each graph to 100M, and vary the average vertex degree in the synthetic graphs from 10 to 80 (i.e., the number of edges changes from 1 billion to 8 billion). For this set of experiments, Figures 5.16(a)–5.16(d) show that the running time of Giraph, GPS, and Pregel+ increases sub-linearly with the number of edges, which indicates that the systems have good scalability (except for GPS which ran out of memory for running PageRank on the two largest graphs). GraphLab has the best scalability among all systems for running PageRank, but it exhibits the worse scalability for running HashMin.

5.5 Conclusions

We evaluated the performance of Giraph [1], GraphLab [30, 16], GPS [40], and Pregel+ [55], w.r.t. various graph characteristics, algorithm categories, optimization techniques, and system scalability. Our results show that, while there is no single system that has superior performance in all cases, Pregel+ and GPS have better overall performance than Giraph and GraphLab. Pregel+ has better performance mainly thanks to its combination of mirroring, message combining, and request-respond techniques, while GPS also benefits significantly from its LALP technique. GraphLab uses vertex-cut partitioning for load balancing, but it incurs extra overhead in locking/unlocking, which may degrade its performance. Giraph generally has the poorer performance because it does not employ any specific technique for handling skewed workload and mainly relies on combiner for message reduction. Finally, we believe that the efficiency of Pregel+ also comes from its C++ implementation, which at least uses less memory, and according to [27] can be 2-3 times faster, than a Java implementation.

Chapter 6

Concluding Remarks

This thesis focuses on the design, implementation and applications in distributed graph computing systems. We consider two types of programming paradigms: “think like a vertex” and “think like a block”. First, we proposed two effective message reduction techniques for Pregel-like systems. Second, we proposed a new graph computing model based on partitioned graph blocks. Third, we identified a set of desirable properties of an efficient Pregel algorithm. Finally, we conducted a comprehensive performance study on existing graph computing systems using graphs with different characteristics and algorithms with different design logics.

Bibliography

[1] Apache giraph. `http://giraph.apache.org/`.

[2] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High speed switch scheduling for local area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, 1993.

[3] D. A. Bader and K. Madduri. GTgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[4] J. Barnat, J. Chaloupka, and J. van de Pol. Improved distributed algorithms for scc decomposition. *Electronic Notes in Theoretical Computer Science*, 198(1):63–77, 2008.

[5] J. Barnat and P. Moravec. Parallel algorithms for finding sccs in implicitly given graphs. In *Formal Methods: Applications and Technology*, pages 316–330. 2007.

[6] 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.

[7] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. M. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *SIGMOD Conference*, pages 1371–1382, 2014.

[8] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. TF-Label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD Conference*, pages 193–204, 2013.

[9] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD Conference*, pages 457–468, 2012.

[10] M. Dayarathna and T. Suzumura. A first view of exedra: a domain-specific language for large graph analytics workflows. In *WWW (Companion Volume)*, pages 509–516, 2013.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] B. Elser and A. Montresor. An evaluation study of BigData frameworks for graph processing. In *BigData Conference*, pages 60–67, 2013.

[13] M. Erwig and F. Hagen. The graph voronoi diagram with applications. *Networks*, 36(3):156–163, 2000.

[14] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[15] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *IPDPS Workshops*, pages 505–511, 2000.

[16] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency - Practice and Experience*, 12(12):1131–1146, 2000.

[17] 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.

[18] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How well do graph-processing platforms perform? An empirical performance evaluation and analysis. In *IPDPS*, pages 395–404, 2014.

[19] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[20] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing pagerank. *Stanford University Technical Report*, 2003.

[21] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: mining radii of large graphs. *TKDD*, 5(2):8:1–8:24, 2011.

[22] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[23] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, pages 938–948, 2010.

[24] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[25] 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.

[26] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson Education India, 2006.

[27] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[28] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *SPAA*, pages 85–94, 2011.

[29] 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.

[30] 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.

[31] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[32] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *CUHK Technical Report (http://www.cse.cuhk.edu.hk/pregelplus/expTR.pdf)*, 2014.

[33] 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.

[34] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT*, pages 331–342, 2011.

[35] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: a fast and scalable tool for data mining in massive graphs. In *KDD*, pages 81–90, 2002.

[36] 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.

[37] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in MapReduce in logarithmic rounds. In *ICDE*, pages 50–61, 2013.

[38] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[39] S. Salihoglu and J. Widom. Computing strongly connected components in pregel-like systems. *Stanford University Tech. Report*.

[40] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.

[41] S. Salihoglu and J. Widom. Optimizing graph algorithms on Pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[42] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD Conference*, pages 979–990, 2014.

[43] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5:269–287, 1983.

[44] Y. Shiloach and U. Vishkin. An O(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[45] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.

[46] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[47] Y. Tao, W. Lin, and X. Xiao. Minimal MapReduce algorithms. In *SIGMOD Conference*, pages 529–540, 2013.

[48] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[49] 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.

[50] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[51] J. C. Wyllie. The complexity of parallel computations. Technical report, Cornell University, 1979.

[52] 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.

[53] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: a resilient distributed graph system on Spark. In *GRADES*, 2013.

[54] 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.

[55] D. Yan, J. Cheng, Y. Lu, and W. Ng. Pregel+: Technical report. *(http://www.cse.cuhk.edu.hk/pregelplus/pregelplus.pdf)*, 2014.

[56] 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.

[57] 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.

[58] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *ScienceCloud*, pages 13–22, 2012.