Pregel+: A Distributed Graph Computing Framework with Effective Message Reduction

[Technical Report]

1. INTRODUCTION

With the popularity of online social networks, mobile communication networks and semantic web services, we have witnessed a growing interest in conducting efficient and effective analysis on these massive graphs. In particular, many distributed graph computing systems have emerged, which are deployed on a shared-nothing distributed computing infrastructure usually built on a cluster of low-cost commodity PCs. Pioneered by Google's Pregel [10], 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 [3], and its open-source implementation Hadoop[1], are another distributed system popularly used for large scale graph processing [5, 8, 6, 14, 11]. 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. The "think like a vertex" programming paradigm also makes it much easier for programmers to design a Pregel algorithm than an equivalent MapReduce algorithm.

Weaknesses of Pregel. Although Pregel's vertex-centric computing model has been widely adopted in most of the recent distributed graph computing systems [9, 1, 12, 7] (and also inspired the edge-centric system [4]), Pregel's vertex-to-vertex message passing mechanism often causes bottleneck in communication when processing real-world graphs. To see this, we first briefly describe how Pregel performs message passing.

In Pregel, a vertex v can send messages to another vertex u only 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 are also Pregel algorithms which require v to send messages to another vertex that is not a neighbor of v. This is common in algorithms that adopt the pointer jumping (or doubling) technique, which is widely used in PRAM algorithms [13].

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 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 impact of 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[1], the maximum degree is over 2.99M while the average degree is only 35. Similarly, in the *BTC* dataset which is a semantic graph converted from an RDF dataset [2], the maximum degree is over 1.6M while the average degree is only 4.69.

We ran *Hash-Min* [11], a distributed algorithm for computing connected components (CCs), over 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* over *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, the 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 [16, 13] for computing CCs, where each vertex v maintains a field $D[v]$ which is the vertex

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

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

Figure 1: Total number of messages sent by each worker, running Hash-Min (with/without mirroring) over the BTC dataset

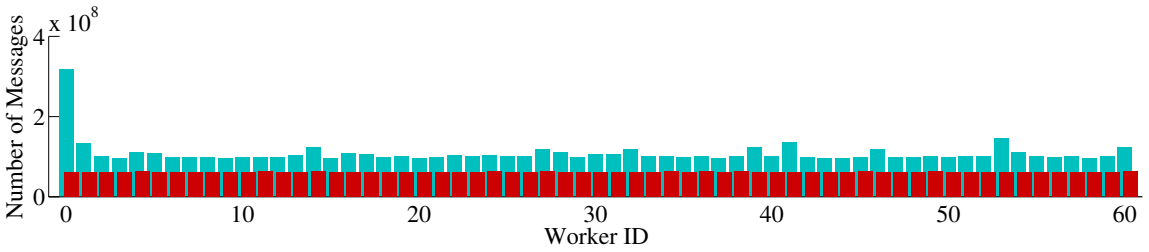

Figure 2: Total number of messages sent by each worker, running S-V (with/without request-respond) over the USA road network

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* over 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 2, the 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. We remark that the imbalanced 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 many more messages than the other workers, as can be observed from Figure 2.

In addition to the two problems mentioned above, Pregel's message passing mechanism is also not efficient for (relatively) dense graphs due to the high overall communication cost. However, graphs like social networks, mobile phone networks, and web graphs are relatively dense, as a person is often connected to at least dozens of people and a web page usually contains many links.

Our solution. In this report, 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, 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 while M is normally from tens to a few hundred. However, as we shall see in Section 4, mirroring a vertex does not always reduce the number of messages due to Pregel's use of message combiner [10]. Hence, we provide a theoretical analysis on what degree a vertex should have in order to be classified as a high-degree vertex, or in other words, which vertices should be selected for mirroring.

In Figure 1, the red bars indicate the total number of messages sent by each 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 (representing Pregel's skewed message passing) and the even-height short red bars (representing balanced communication load in Pregel+ 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 improves the runtime of *Hash-Min* over BTC from 27.91 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 caused by 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. Again, when dealing with large real-world graphs, ℓ is up to orders of magnitude greater than M.

In Figure 2, the 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 even-height short red bars. In particular, Vertex 0 now only responds to all the requesting workers in the last round, instead of all the requesting vertices, and hence the highly imbal-

anced workload caused by Vertex 0 in Worker 0 is now evened out. The request-respond paradigm improves the runtime of *S-V* over the USA road network from 261.9 seconds to 137.7 seconds.

Finally, we remark that our two message reduction techniques incur slightly more computational overhead than the basic Pregel framework, but greatly reduce the communication cost. As the communication cost is often the dominant cost, our techniques are shown to significantly improve the overall performance of Pregel algorithms. In fact, the communication cost is more sensitive to the network condition. For example, Pregel algorithms run in a public data center are expected to benefit more from our message reduction techniques than if they were run in a cluster without any resource contention such as the one used in our experiments.

2. BACKGROUND AND RELATED WORK

In this section, we first briefly review the Pregel framework, and present some representative Pregel algorithms which will be helpful in later sections when we introduce our Pregel+ system. We then discuss other related distributed graph computing systems.

2.1 Pregel

Pregel [10] 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. In the following we describe the Pregel algorithms for three typical applications, which will be used for illustration throughout the report.

We first define the graph notations used in the report. 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 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 δ.

Application 1: Connected Components (CCs). The first application is to compute all the connected components in an undirected graph. We adopt the *Hash-Min* algorithm [11]. 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 $v \in V$.

We now describe the *Hash-Min* algorithm. 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 $min(v) = cc(v)$.

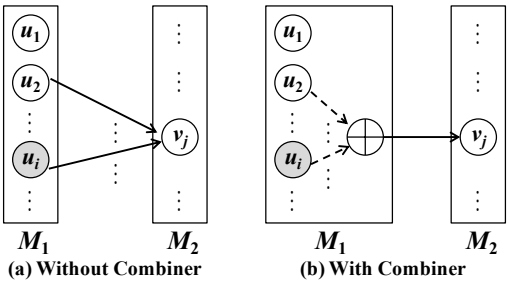

Figure 3: Illustration of Combiner

Application 2: 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, $pr(v)$, of each vertex $v \in V$. Pregel's PageRank algorithm [10] works as follows. In superstep 1, each vertex v initializes $pr(v) = 1/|V|$ and distributes the value $(pr(v)/|\Gamma_{out}(v)|)$ 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 $(pr(v)/|\Gamma_{out}(v)|)$ to each of its out-neighbors.

Application 3: 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 algorithms such as that of computing bi-connected components [16]. 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 $a(u)$; 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 $\widehat{\Gamma}_{out}(v)$.

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 algorithm mentioned above, 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. Figure 3 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. Similarly, in *Hash-Min*, messages to the same target vertex can be combined into a single message that has the smallest value among these messages.

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.

2.2 Pregel-Like Systems in JAVA

Since Google's Pregel is proprietary, many open-source Pregel counterparts are developed, such as Giraph [1] and GPS [12]. These systems are implemented in JAVA to be compatible with Hadoop, as they all 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 should be implemented in a language where programmers manage main memory objects themselves, such as C/C++. 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 [12] supports an optimization called large adjacency list partitioning (LALP) to handle high-degree vertices. However, GPS does not perform sender-side message combining, as they claim that very small performance difference can be observed whether it is used or not. A recent system, Giraph++ [15], extends Giraph to support a graph-centric programming paradigm that opens up the partition structure (i.e., all the vertices on a machine) to users. In Giraph++, each partition is treated as two sets of vertices, internal ones and boundary ones. Here, an internal vertex is like a primary vertex copy in Pregel+, while an external vertex is like a mirror. However, Giraph++'s use of mirrors is not much different than the use of a combiner: a combined message to a target vertex v is now replaced by the state of v's mirror on the local machine. Giraph++'s mirroring does not eliminate bottleneck vertices with high degree, but incurs excessive space cost as mirrors are constructed even for low-degree vertices.

2.3 GraphLab and PowerGraph

GraphLab [9] is another vertex-centric parallel graph computing system but with 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 far as we know, Pregel+ is the only distributed graph computing system that supports both data pulling and data pushing.

GraphLab also builds mirrors for vertices, which are called ghosts. Since asynchronous execution cannot support algorithms that perform different operations by branching on the superstep number, GraphLab also provides support for synchronous execution. However, GraphLab does not support combiner, while Pregel+ supports both vertex mirroring and message combining. GraphLab also creates ghosts for every vertex regardless of their degree, which leads to excessive space consumption.

There is a recent version of GraphLab called PowerGraph [4], which 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, and vertex values must be updated by a commutative and associative operation. This significantly reduces the expressiveness of GraphLab. For example, the attribute-broadcast problem described in Section 2.1 has a straightforward Pregel algorithm, but it is non-trivial to solve the problem with Graphlab's GAS model.

3. SYSTEM OVERVIEW OF PREGEL+

We now give an overview of the Pregel+ system, particularly the programming interface and communication framework of Pregel+. We focus our discussion on the basic vertex-centric computing functionalities in this section, before we present the system extensions to support effective message reduction in the next two sections.

3.1 Programming Interface

An important feature of a graph-parallel abstraction is its user-friendliness. Similar to Pregel, writing a Pregel+ program involves subclassing a group of predefined classes, with the template arguments of each base class properly specified. Pregel+ provides the following two base classes for programmers.

- ***Vertex<I, V, E, M, H>***. An object of the *Vertex* class contains four fields: (1)a vertex ID of type *<I>*; (2)a value field of type *<V>*; (3)an adjacency list of type *<E>*; and (4)a boolean state indicating whether the vertex is active. Template argument *<M>* specifies the message type, while *<H>* specifies the hash function that takes the ID of a vertex as input and computes the machine ID for the vertex (i.e., the machine in which the vertex resides). If *<H>* is not specified, a default hash function is used. The *Vertex* class also has an abstract *compute*() function that takes a set of messages of type *<M>* as input. A user-defined subclass of *Vertex* needs to implement the *compute*() function.

- ***Worker<VertexT>***. We use the term "worker" to represent a computing unit, which can be a machine or a thread/process in a machine. For simplicity of discussion, we assume that each machine runs only one worker but the concepts can be straightforwardly generalized. The *Worker* class has three functions: (1)*load_vertex*(*line*), which specifies how to parse a line from the input file into a vertex object; (2)*dump_vertex*(*vertex*, *HDFS_writer*), which specifies how to dump a vertex object to the output file in HDFS; (3)*run()*: which implements the graph computation and is called by users to run the specified job.

To illustrate, we show how to implement *Hash-Min* in Pregel+. We first decide the template arguments of the *Vertex* class. In *Hash-Min*, each vertex v has an integer ID, and an integer field $min(v)$ indicating the smallest vertex ID seen, and sends $min(v)$ as a message to other vertices. Thus, *<I>*, *<V>*, *<E>* and *<M>* are all specified as integer type. We then define a class *CCVertex* that inherits this *Vertex* class, and implement the *compute*() function using the *Hash-Min* algorithm described in Section 2.1. To run the *Hash-Min* job, we inherit the *Worker<CCVertex>* class, implement the vertex loading/dumping functions, and then call *run*() to start the job.

3.2 Communication Framework

In Pregel+, each worker is simply an MPI (Message Passing Interface) process and communications among different processes are

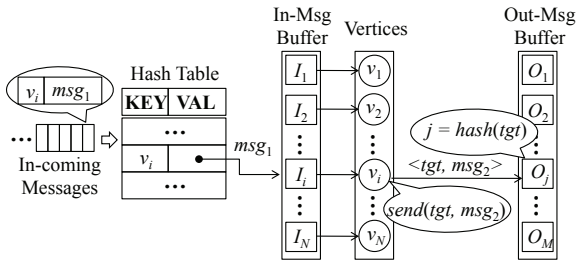

Figure 4: Illustration of Message Channel, Ch_{msg}

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}. When a worker has called *compute*() for all its active vertices, the messages in Ch_{msg} are sent to the target workers 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. As far as we know, no previous work has explored different designs of the message channel and compared their performance. We describe three different options of implementing Ch_{msg}, which are described as follows.

Option 1: the first approach is presented in Figure 4, where 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 at 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 operation occurs, where updates to T_{in} is 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*, 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 **<H>** specifies, then the target vertex is in machine $M_{hash(tgt)}$. Thus, msg_2 (along with tgt) is appended to the end of the buffer $O_{hash(tgt)}$. After *compute*() is called for all active vertices, the messages in each buffer O_j are then sent to each worker M_j. 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.

Option 2: the second approach makes the following changes to the first approach. It groups the messages in an outgoing message buffer by target vertex IDs even if no combiner is used, so that the target ID is associated with each group rather than each message, hence reducing the amount of communication as the target vertex IDs are not sent multiple times when there are more than one message for the same target vertex. When receiving messages, a worker simply merges the messages sent from different workers in a merge-sort style. Meanwhile, a worker maintains its vertices in sorted order of vertex IDs. Since the messages are sorted, it is not necessary to maintain T_{in} and I_i.

Option 3: the third approach makes the following changes to the first approach. It organizes an outgoing message buffer as a hash table, with target ID tgt as the key and the set of messages (or the currently combined message) directed to tgt as the value, instead of a list of $\langle tgt, msg \rangle$ entries. There is no need to sort the messages in O_i in this case.

We now analyze the differences between the three approaches. The main difference among them is the way how outgoing messages are handled: Option 1 groups messages by sorting only when a combiner is applicable; Option 2 always groups messages by sorting; Option 3 groups messages by hashing. Note that when a combiner is used, the first approach handles outgoing messages in the same way as the second approach; in this case, they differ by the way they handle incoming messages.

3.3 Pregel+ Implementation

We make Pregel+ open-source. All the system source codes, as well as the source codes of the applications discussed in this paper, 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.

Unlike Pregel, Pregel+ also makes the master a worker, and the task of fault recovery can be implemented by a script as follows. The script loops a Pregel+ 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.

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 5 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 on M_2 and k neighbors on M_3. Figure 5(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 5(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 5(b). In total, only two messages are sent through the network, which is not only a tremendous saving in 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 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 $a(v)$*

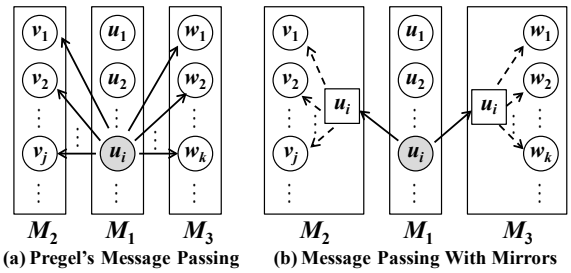

Figure 5: Illustration of Mirroring

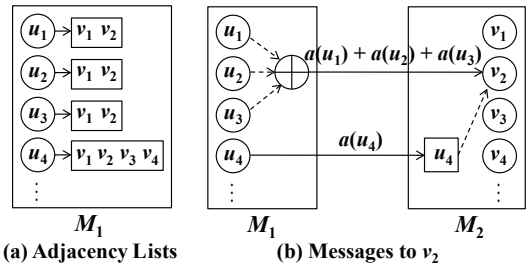

Figure 6: Mirroring v.s. Message Combining

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. ■

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 $d(v) \geq \tau$. 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 sender vertex.

Figure 6 illustrates the concepts of Ch_{msg} and Ch_{mir}, where the focus is on 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 6(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 6(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 and 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 6(b), when u_4 in M_1 sends $a(u_4)$ to its mirror in M_2, it 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 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.

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 3). The message value $a(v)$ is then forwarded to the incoming message buffers of v's neighbors locally in M_i.

Programming Interface. Our Pregel+ system provides two classes, *MVertex* and *MWorker*, for coding programs that applies mirroring. There are only 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 $u \in \Gamma_{out}(v)$. In fact, the ID of the neighbor u may not be available if v uses mirroring.

Referring to the applications described in Section 2.1 again. For PageRank computation, 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 value of type <*V*>, but also decided by the edge (i.e., adjacency list item) of type <*E*>. For example, In Pregel's algorithm for single-source shortest path (SSSP) computation [10], 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 an object of type <*E*> 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. For example, in Figure 6, *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 user-specified type <*E*> should overload the function *relay*(msg) to add the edge length $\ell(v, u)$ to msg, which updates the value of msg to the desired value $(d(v) + \ell(v, u))$.

5. THE REQUEST-RESPOND PARADIGM

Our second message reduction technique is to add a new request-

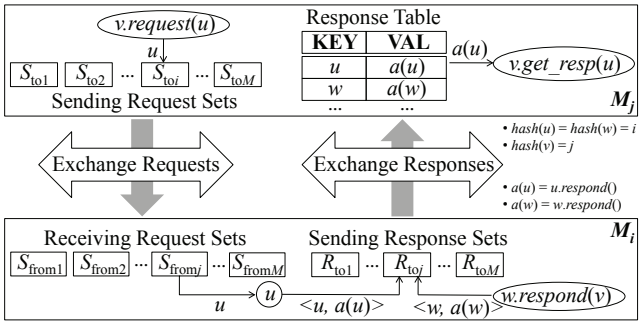

Figure 7: Illustration of the Request-Respond Paradigm

respond functionality to the basic Pregel framework. We first illustrate the concept by using the application of attribute broadcast presented in Section 2.1.

While the Pregel algorithm for attribute broadcast presented in Section 2.1 requires 3 supersteps and is cumbersome to implement, with the new request-respond paradigm, attribute broadcast 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 $\widehat{\Gamma}_{out}(v)$.

Request-Respond Message Channel. We now explain in detail how Pregel+ supports the request-respond paradigm. 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 7 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 on machine M_k. In a superstep, a vertex v in a 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 7, 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 7, 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. *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 framework. In practice, however, there are often some vertices (i.e., the bottleneck vertices described in Section 1) 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. 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 $\langle w, w.respond()\rangle$ to the response set $R_{\text{to}\,j}$ where $j = hash(v)$. This process is also illustrated in Figure 7. 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 over a 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; 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. Pregel+ provides two classes, *RVertex* and *RWorker*, for programming with the request-respond paradigm. Compared with the *Vertex* class, *RVertex* has one more template argument **<*R*>** to be specified, 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.

Computational Overhead. There is overhead incurred by request/response exchange, but this cost is also required if we use Pregel's

vertex-to-vertex message passing to obtain the same result. There is only one extra cost in the request-respond paradigm, which is the use of table lookup to get the attribute value of another vertex, while Pregel only requires checking the incoming messages sent to the current vertex. However, this overhead is insignificant compared with the reduction in communication cost achieved by the request-respond paradigm

6. REFERENCES

[1] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

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

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

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

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

[6] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[7] 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.

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

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[10] 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.

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

[12] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, page 22, 2013.

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

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

[15] 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.

[16] D. Yan, J. Cheng, Y. Lu, and W. Ng. Practical pregel algorithms for massive graphs. *Technical Report (http://www.cse.cuhk.edu.hk/pregelplus/papers/ppa.pdf)*, 2013.