GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit

Da Yan[1], Yuzhen Huang[2], Miao Liu[3], Hongzhi Chen[4], James Cheng[5], Huanhuan Wu[6], Chengcui Zhang[7]

Abstract—We propose GraphD, an out-of-core Pregel-like system targeting efficient big graph processing with a small cluster of commodity PCs connected by Gigabit Ethernet, an environment affordable to most users. This is in contrast to some recent efforts for out-of-core graph computation with specialized hardware. In our setting, a vertex-centric program is often data-intensive, since the CPU cost of calculating a message value is negligible compared with the network cost of transmitting that message. As a result, network bandwidth is usually the bottleneck, and out-of-core execution would not sacrifice performance if disk IO overhead can be hidden by message transmission, which is achieved by GraphD through the parallelism of computation and communication. GraphD streams edge and message data on local disks, and thus consumes negligible memory space. For a broad class of Pregel algorithms where message combiner is applicable, GraphD completely eliminates the need of any expensive external-memory join or group-by, which is required by existing systems such as Pregelix and Chaos. Extensive experiments show that GraphD beats existing out-of-core systems by orders of magnitude, and achieves comparable performance to in-memory systems running with adequate memory.

Index Terms—Out-of-core, graph, vertex-centric, Pregel.

1 Introduction

Since the advent of Pregel [15], various vertex-centric systems have been actively developed for processing big graphs [4], [13], [6], [1]. In these systems, a programmer only needs to specify the behavior of one generic vertex when developing distributed graph algorithms. Due to this user-friendly programming model, and good horizontal scalability, vertex-centric systems have been popularly used in various real applications such as social network analysis [19] and graph matching [5], [23].

However, most distributed vertex-centric systems require the entire input graph to reside in main memory of machines. Intermediate data generated for communication among machines are also buffered in memory, and the space consumption can be very high. For example, [33] reported that to process a graph dataset that takes only 28GB disk space, Giraph and GraphLab need 370GB and 800GB memory space, respectively.

While memory is becoming cheaper and memory-rich clusters are becoming affordable to big companies and well-funded research labs, this is still not the case for many small businesses and researchers. For example, [1] reported that in the Giraph user mailing list there are 26 cases of out-of-memory related issues from March 2013 to March 2014. Nevertheless, it is often the large body of small businesses and researchers who have urgent need of scalable graph processing technologies, while big companies have the capability of developing their own proprietary systems.

Modern applications often generate very big graphs, such as online social networks, the Web graph, and knowledge graphs. However, to perform in-memory PageRank computation over a Twitter graph with 1.96 billion follow-edges (see Table 1 in Section 6), Giraph and Pregel+ [30] need nearly 264GB and 109GB memory space in our cluster, respectively. To process a Web graph like *ClueWeb* with 42.6 billion edges (see Table 1), Giraph and Pregel+ [30] would need 5.7TB and 2.4TB memory space, respectively, which is prohibitive.

To process a big graph beyond the memory limit, several out-of-core systems were developed for running with the disk of a single machine (often a PC), such as GraphChi [12], X-Stream [21] and VENUS [3]. For small graphs, these systems may beat a distributed one since there is no expensive network communication. However, such a system needs to stream and process the entire graph, and the execution time increases with the graph size due to fixed disk bandwidth; this is in contrast to a distributed system where each machine only needs to process a portion of the input graph. As a result, distributed out-of-core systems such as Pregelix [1] and Chaos [20] were recently developed for streaming the disks of all machines concurrently, to achieve high aggregate disk bandwidth. As confirmed by our experiments in Section 6, distributed systems can be much more efficient than single-machine ones when processing a big graph.

To compensate for the low disk bandwidth in a standalone environment, researchers have explored the potential of utilizing flash memory as the external memory media. Flash memory supports significantly faster random access time than magnetic disks, and can service multiple concurrent IO requests. Systems like FlashGraph [32] and G-Store [11] use multithreading to achieve maximum IO performance out of flash memory. However, large flash memory is still not widely available to every small business and researcher for the time being.

We target the setting of a small cluster of commodity PCs connected by Gigabit Ethernet, which is affordable to most users. In this environment, disk streaming bandwidth is usually much higher than network bandwidth, and network communication is usually the performance bottleneck of a vertex-centric program. Specifically, vertices communicate by message passing, and the CPU cost of calculating a message value is negligible compared

• *Da Yan and Chengcui Zhang are with the Department of Computer Science, the University of Alabama at Birmingham. The other authors are with the Department of Computer Science and Engineering, the Chinese University of Hong Kong.*
E-mails: {[1]yanda, [7]czhang02}@uab.edu, {[2]yzhuang, [3]mliu, [4]hzchen, [5]jcheng, [6]hhwu}@cse.cuhk.edu.hk

with the network cost of transmitting that message. In our targeted setting, a distributed vertex-centric system does not need to keep graph and messages in memory: they can be streamed on disks, and as long as the disk IO cost is hidden by the communication cost, scalability is achieved without sacrificing performance.

In this paper, we introduce our out-of-core Pregel-like system, GraphD, for efficient big graph processing on a small cluster of commodity PCs connected by Gigabit Ethernet. We remark that GraphD is for use when the aggregate memory space is insufficient for in-memory processing; otherwise, one may use an existing in-memory Pregel-like system instead. Also, GraphD is designed for the normal setting without special hardware. If large flash memory is deployed, one may use dedicated systems like FlashGraph [32] or G-Store [11]; while if high-speed network is available, one may use dedicated systems like Chaos [20] or GraM [27] .

GraphD provides the following desirable features:

- **Bounded Memory Space.** When a graph $G = (V, E)$ is processed with n machines, we prove that each machine only requires $O(\frac{|V|}{n})$ memory space.
- **Efficient Sparse Computation.** GraphD automatically adapts the amount of edges streamed from local disks to the number of active vertices that perform computation.
- **Overlapping Disk and Network IO.** GraphD buffers outgoing messages to local disks to reduce memory consumption, and this cost is hidden by the slower message transmission that executes in parallel.
- **Efficient External-Memory Processing.** For a broad class of Pregel algorithms where message combiner is applicable, GraphD uses a technique called ID-recoding to eliminate the need of any expensive external-memory join or group-by, as required by other systems like Pregelix.

Extensive experiments demonstrate that GraphD is able to achieve comparable performance to an in-memory Pregel-like system.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the execution model of GraphD, and analyzes its space cost. Section 4 discusses the parallel framework of GraphD which fully overlaps computation with communication. Section 5 describes the ID recoding technique and Section 6 reports experimental results. Finally, the paper is concluded in Section 7.

2 Background and Related Work

We first review the computation model of Pregel, and then introduce other vertex-centric systems for processing big graphs.

In this paper, we assume that the input graph $G = (V, E)$ is stored on Hadoop Distributed File System (HDFS), where each vertex $v \in V$ has a unique ID $id(v)$ and an adjacency list $\Gamma(v)$. For simplicity, we use v and $id(v)$ interchangeably. If G is undirected, $\Gamma(v)$ contains all v's neighbors; while if G is directed, $\Gamma(v)$ contains all v's out-neighbors. The degree (or out-degree) of v is denoted by $d(v) = |\Gamma(v)|$. Each vertex v also maintains a value $a(v)$ which gets updated during computation. A Pregel program is run on a cluster of machines, $\mathbb{W}$, deployed with HDFS.

2.1 Pregel Review

Computation Model. A Pregel program starts by loading an input graph from HDFS into the memory of all machines. Each vertex v is distributed to a machine $W = hash(v)$ along with $\Gamma(v)$, where $hash(.)$ is a partitioning function that takes vertex ID as the input. We denote the set of all vertices that get assigned to W by $V(W)$. Each vertex v also maintains a boolean field $active(v)$ indicating whether v is active.

A Pregel program proceeds in iterations, where an iteration is called a superstep. In Pregel, a user needs to specify a user-defined function (UDF) *compute*(*msgs*) to be called by a vertex v, where *msgs* is the set of incoming messages received by v (sent in the previous superstep). In v.*compute*(.), v may update its value $a(v)$, send messages to other vertices, and vote to halt (i.e., deactivate itself). Only active vertices will call *compute*(.) in a superstep, but a halted vertex will be reactivated if it receives a message. The program terminates when all vertices are halted and there is no pending message towards the next superstep. The final results are dumped to HDFS at last.

To illustrate how to write *compute*(.), we consider the PageRank algorithm of [15] where $a(v)$ stores the PageRank value of vertex v, and $a(v)$ gets updated until convergence. In Step 1, each vertex v initializes $a(v) = 1/|V|$ and distributes $a(v)$ to its out-neighbors by sending each out-neighbor a message $a(v)/d(v)$. In Step i ($i > 1$), each vertex v sums up the received message values, denoted by sum, and computes $a(v) = 0.15/|V| + 0.85 \cdot sum$. It then distributes $a(v)/d(v)$ to each of its out-neighbors.

To reduce communication, users may implement a message combiner to specify how to combine messages targeted at the same vertex v_t, so that messages generated on a machine W towards v_t will be combined into a single message by W locally, and then sent to v_t. For example, in PageRank computation, the combiner can be implemented as computing sum, since only the sum of incoming messages is of interest in *compute*(.).

Pregel also allows users to implement an aggregator for global communication. Each vertex can provide a value to an aggregator in *compute*(.) in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep.

In an in-memory Pregel-like system, for each vertex v, machine $W = hash(v)$ keeps the following information in main memory: (1) the vertex state, which consists of $id(v)$, $a(v)$ and $active(v)$, and (2) the adjacency list $\Gamma(v)$. As vertex degree is used by a few existing out-of-core systems (e.g., GraphChi [12], VENUS [3], FlashGraph [32]) to demarcate the adjacency lists of different vertices, for the consistency of presentation, we also include $d(v)$ into the vertex state of v, which is given as follow:

$$state(v) = (id(v), a(v), active(v), d(v)). \quad (1)$$

2.2 Vertex-Centric Systems

Besides Pregel, many other vertex-centric graph processing systems have been developed. We categorized into three classes (1) distributed in-memory systems, and (2) single-machine out-of-core systems, and (3) distributed out-of-core systems. For (1) and (3), network bandwidth is the bottleneck, and there are dedicated systems to improve communication throughput using high speed network. For (2), the disk bandwidth is the bottleneck, and there are dedicated systems to improve IO throughput using flash memory. We now review each category of systems, and explain why they are insufficient for the setting that GraphD targets.

Distributed In-Memory Systems. Since Pregel [15] is only for internal use in Google, many open-source Pregel-like systems emerge including Giraph [4], Pregel+ [30], GraphX [7] and GPS [22]. Like Pregel, these systems keep an entire input

graph in memory during computation, and also buffer intermediate messages in memory. However, network communication is usually the performance bottleneck rather than CPUs, and thus GraM [27] utilizes RDMA-based communication over Infiniband to greatly improve the network bandwidth, allowing communication to overlap with computation to preserve the multi-core parallelism.

Unlike Pregel that adopts synchronous execution where vertex communicates by message passing, GraphLab [13], [6] adopts a shared-memory abstraction where a vertex directly pulls data from its adjacent vertices/edges, and asynchronous execution is supported to allow faster convergence for algorithms where vertex values converge asymmetrically. Since this work focuses on out-of-core systems, we refer interested readers to [8], [14] for more discussions on existing in-memory vertex-centric systems.

Recently, [25] developed tailor-made graph analytics programs using MPI and OpenMP, which scale up to thousands of nodes in the NCSA *Blue Waters* supercomputer, and outperform existing vertex-centric systems on a small cluster. This demonstrates a tradeoff between user-friendliness and efficiency of execution.

Single-Machine Out-of-Core Systems. These systems partition vertices according to disjoint ranges of vertex ID, and load one vertex partition to memory at a time for processing. GraphChi [12] needs to load all vertices in a partition, along with all their adjacent edges, into memory before processing of the partition begins. X-Stream [21] only loads all vertices in a partition into memory, while edges are streamed on local disk. In both GraphChi and X-Stream, a vertex communicates with each other by writing/reading data on adjacent edges. VENUS [3] avoids the cost of writing data to edges, by letting a vertex obtain values directly from its in-neighbors. However, VENUS is not open source.

While X-Stream does not require edges for a partition to be pre-sorted like the other systems do, it needs to scan every edges on disk in each iteration, even if only a small number of vertices require computation. Unfortunately, this inefficiency for sparse computation workload is also inherited by its scale-out version, Chaos [20]. In contrast, GraphChi and VENUS support *selective scheduling* which skips scanning those vertex partitions that do not contain active vertices, but the effectiveness is limited since a partition needs to be scanned even if it contains only one active vertex. As we shall see in Section 3.2, GraphD provides an elegant approach to adapt disk IO cost to the workload sparsity.

In the above systems, disk bandwidth is the performance bottleneck. While GraphChi targets PC environment, X-Stream also considers the setting of a high-end server with many cores, and improves parallelism by streaming edges on flash memory which provides higher IO bandwidth. However, the key features of flash memory, i.e., significantly faster random access than magnetic disk, and the capability of serving multiple concurrent and asynchronous IO requests, are not utilized. Dedicated systems like FlashGraph [32] and G-Store [11], and dedicated approaches like [18] thus use multithreading to fully exploit the bandwidth of flash memory and thus better utilizing the many cores.

Like [25], [16] noticed that vertex-centric frameworks gain user-friendliness at the cost of reduced performance, and a tailor-made program can run much faster using the SSD (or even just memory) of a laptop. There are also systems for processing big graphs in a shared-memory environment, such as Ligra [24] and Galois [17], which demonstrate superb performance due to high parallelism, but require a machine with big RAM (e.g., 1TB).

Distributed Out-of-Core Systems. Compared with single-machine systems, these systems only require each machine to process a partition of the graph, and the disk bandwidth of all machines are utilized in parallel. HaLoop [2] improves the performance of Hadoop for iterative computation, by allowing a job to cache data to local disks to avoid remote reads. However, HaLoop still adopts the MapReduce model rather than the user-friendly vertex-centric model. Pregelix [1] formulates Pregel's computation model using relational operators like join and group-by, which are relatively expensive. It then leverages a general-purpose dataflow engine for execution. Giraph also supports out-of-core execution, but [1] reported that it does not function properly.

Chaos [20] scales out X-Stream in order to use the aggregated disk bandwidth in a cluster. Edge streaming workloads are distributed among multiple machines, and work stealing is adopted for load balancing. Chaos' design allows high performance when machines in the cluster are connected by high-speed network (e.g., 40GigE). This is because Chaos spreads data over the machines, managed by a storage sub-system, and every computing thread requests the necessary data for processing from the storage subsystem in the unit of chunks to allow sequential data access. The data transmission cost, however, is not small when the network speed (e.g., GigE) is not high enough. Thus, as reported in [20], Chaos' performance was undesirable using GigE, which is typically used in the type of clusters GraphD is designed for.

3 Data Organization and Streams

In this section, we describe the distributed semi-streaming (DSS) model of GraphD, show that its memory cost is $O(|V|/|\mathbb{W}|)$ and introduce the design of its disk-resident streams.

3.1 Distributed Semi-Streaming Model

First consider Pregel's memory cost. For simplicty, we assume that the types of vertex ID, vertex value, adjacency list item, and message all have constant size[1]. Since $active(v)$ and $d(v)$ also have constant size, $state(v)$ defined in Eq (1) has constant size.

Recall that Pregel keeps the $O(|V|)$ vertex states and $O(|E|)$ adjacency list items in memory. Let us denote the set of messages currently in the system by M, where a message is buffered either on the sender-side or on the receiver-side. Then, $O(|M|)$ memory space is also required for keeping messages. Therefore, the total memory space required by Pregel is $O(|V| + |E| + |M|)$.

Usually, $O(|E|)$ is much larger than $O(|V|)$. For example, a user in a social network can easily have tens of friends. Also, $O(|M|)$ is large. For example, in PageRank computation, one message is transmitted along each edge in a superstep, and thus $O(|M|) = O(|E|)$; while in the triangle finding algorithm of [19], a superstep can transmit up to $O(|E|^{1.5})$ messages. Therefore, the dominating memory cost is contributed by adjacency lists (i.e., $O(|E|)$) and messages (i.e., $O(|M|)$). GraphD streams adjacency lists and messages on local disks, keeping only the $O(|V|)$ vertex states in memory. We remark that even the $O(|V|)$ vertex states can be kept on local disks, as a vertex can be streamed along with its adjacent edges for vertex-centric computation at each time; however, as we shall see in Section 3.2, maintaining vertex states in memory allows GraphD to skip reading the edges of inactive vertices, which is important when computation workload is sparse.

The $O(|V|)$ vertex states may still be too large to fit in the memory of a single machine. Since GraphD is a distributed

1. These data types are specified by users through C++ template arguments, and can have variable sizes in reality (e.g., vertex ID can be a string)

system, each machine only needs to keep a portion of vertex states. GraphD follows a model called *distributed semi-streaming* (DSS)[2], where each machine W only keeps the states of all its assigned vertices, $V(W)$, in its memory, and treats their adjacency lists and incoming and outgoing messages as local disk streams.

It remains to show that DSS distributes the vertex states evenly among the $|\mathbb{W}|$ machines, i.e., each machine holds no more than $O(\frac{|V|}{|\mathbb{W}|})$ vertex states with a small constant (e.g., 2). We now prove this property, regarding the machine number $|\mathbb{W}|$ as a constant.

Lemma 1. *Assume that $hash(.)$ is well chosen so that a vertex is assigned to every machine with equal probability, then with probability of at least $1-O(\frac{1}{|V|})$, it holds that $\max_{W\in\mathbb{W}}|V(W)|$ is less than $2\frac{|V|}{|\mathbb{W}|}$.*

Proof. First, consider a particular machine W. Since every vertex is hashed to W with probability $p = \frac{1}{|\mathbb{W}|}$, the total number of vertices that are hashed to W (i.e., $|V(W)|$) conforms to a binomial distribution with mean $\mu = |V|p$ and variance $\sigma^2 = |V|p(1-p) < |V|p = \mu$.

According to Chebyshev's inequality, we have

$$\Pr\left(\Big||V(W)| - \mu\Big| \geq \mu\right) \leq \sigma^2/\mu^2.$$

Since $\sigma^2 < \mu$, the R.H.S. is less than $1/\mu$. Moreover, the L.H.S. is at least $\Pr(|V(W)| \geq 2\mu)$. Therefore, we obtain

$$\Pr(|V(W)| \geq 2\mu) < 1/\mu. \tag{2}$$

Since $\mu = \frac{|V|}{|\mathbb{W}|}$, $\frac{1}{\mu} = \frac{|\mathbb{W}|}{|V|} = O(\frac{1}{|V|})$ is a very small number. For example, when we process a billion-node graph using a cluster of 20 machines, $|\mathbb{W}|$ is only 20 but $|V|$ is in the order of 10^9, and thus $1/\mu$ is in the order of 10^{-7}–10^{-8}.

We now consider all machines in $\mathbb{W}$, and proceed to prove our lemma:

$$\begin{aligned}
& \Pr(\max_{W\in\mathbb{W}}|V(W)| < 2\frac{|V|}{|\mathbb{W}|}) \\
= \;& \Pr(\max_{W\in\mathbb{W}}|V(W)| < 2\mu) \\
= \;& \Pr\Big(\bigwedge_{W\in\mathbb{W}}\Big\{|V(W)| < 2\mu\Big\}\Big) \\
\geq \;& 1 - \sum_{W\in\mathbb{W}}\Pr(|V(W)| \geq 2\mu) \quad \text{(using union bound)} \\
> \;& 1 - \frac{|\mathbb{W}|}{\mu} \quad \text{(using Eq (2)).}
\end{aligned}$$

The lemma is proved by noticing that $\frac{|\mathbb{W}|}{\mu} = \frac{|\mathbb{W}|^2}{|V|} = O(\frac{1}{|V|})$. For example, when $|\mathbb{W}|$ is 20 and $|V|$ is in the order of 10^9, $\frac{|\mathbb{W}|^2}{|V|}$ is in the order of 10^{-6}–10^{-7}. □

We additionally require that the main memory of a machine be large enough to hold the state $state(v)$ and adjacency list $\Gamma(v)$ of any single vertex v, so that v can access them in v.*compute*(.). We add this constraint because $\Gamma(v)$ of a high-degree vertex v could require more memory space than $O(\frac{|V|}{|\mathbb{W}|})$ (i.e., the bound of Lemma 1), but this constraint is reasonable given the RAM size of a commodity PC today, and it is also required by existing out-of-core systems such as GraphChi and Pregelix.

2. We name the model as DSS due to its similarity to the semi-streaming computation of external-memory graph algorithms.

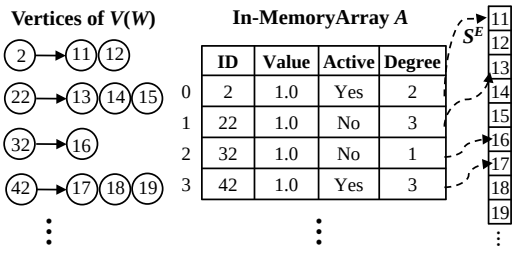


Fig. 1. Vertex States and Edge Stream of a Machine W

3.2 Graph Organization and Edge Streams

While GraphD may load data from and write results to HDFS, during iterative computation, GraphD only sequentially reads/writes binary streams on local disks for efficiency. In GraphD, a stream is implemented as a file, which is sequentially read (or appended) using an in-memory buffer B of size b, and B is refilled (or flushed) when its end is reached. We set $\mathbf{b = 64\ KB}$ which is empirically tested to be sufficient to exhibit sequential IO.

When users specify GraphD to load an input graph from HDFS, the vertices get partitioned among all machines like in Pregel (recall Section 2.1), where each machine W saves the adjacency lists of its assigned vertices, $V(W)$, to its local disk as an edge stream, denoted by S^E. Meanwhile, the states of the assigned vertices, $V(W)$, are kept in memory (for computation) and also written to local disk (for subsequent local loading, see below). Optionally, if the graph was previously loaded from HDFS by another job, users may also specify GraphD to let each machine directly load the previously saved vertex states to memory.

In GraphD, each machine organizes its in-memory vertex states with an array A, as illustrated in Figure 1. Vertices in A are ordered by vertex ID (e.g., 2, 22, 32, 42, $\cdots$ in Figure 1), and the edge stream S^E simply concatenates their adjacency lists in the same order. In a superstep, *compute*(.) is scheduled to be called on the active vertices in A in order. Since a vertex v needs to access $\Gamma(v)$ in v.*compute*(.), the next $d(v)$ items are sequentially read from S^E to form $\Gamma(v)$. Thus, each superstep only sequentially reads S^E once. If topology mutation is enabled, each superstep would digest an old edge stream and generate a new edge stream.

Naïve streaming of an entire edge stream is inefficient if only a small number of vertices are active. For example, X-Stream adopts this method and [21] admitted that X-Stream is inefficient for *graphs whose structure requires a large number of iterations*.

In Figure 1, the edges of inactive vertices 22 and 32 can actually be skipped if they receive no message. To implement this idea efficiently, GraphD supports a function *skip*(k), which skips the next k items from the stream. Referring to Figure 1 again, after vertex 2 is processed, we may skip the edges of vertices 22 and 32 by calling *skip*(4), where 4 is computed by adding their degrees $d(v)$ (i.e., 3 and 1 in array A). However, it is inefficient to perform a random disk read for each time *skip*(.) is called, especially when there are many small series of inactive vertices in A: the many random reads could be more costly than streaming the whole S^E.

We want to skip a long series of inactive vertices with a random read, but still achieve sequential disk bandwidth in dense workloads. We now describe how we achieve this goal. To achieve this goal, when streaming S^E, *skip*(.) avoids reading data if after the skipping, the position to read data from is still in the stream buffer B; otherwise, B is refilled starting from the new read-position. Obviously, this approach limits the number of random reads to be at most that incurred when streaming the whole S^E.

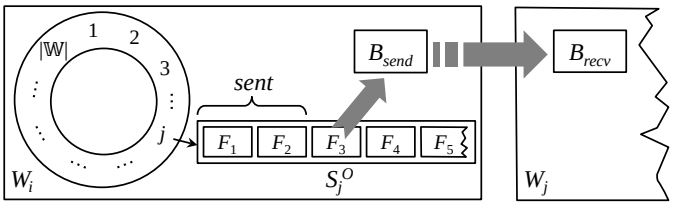


Fig. 2. Sending Messages in OMSs

3.3 Message Streams

Overview. Each machine maintains multiple edges streams on its local disk, including one *incoming message stream* (IMS), denoted by S^I; and $|\mathbb{W}|$ *outgoing message streams* (OMSs) S_i^O $(i = 1, 2, \cdots, |\mathbb{W}|)$, where each OMS S_i^O is used to buffer those messages destined at vertices on the i-th machine, denoted by W_i. When a vertex v sends a message to another vertex u in v.*compute*(.), we append the message to OMS $S_{hash(u)}^O$.

To overlap computation with communication, we require each OMS to support concurrent data appending (at the tail) and data fetching (at the head). This is because messages are constantly generated by vertex-centric computation in high velocity, and have to be buffered into disk streams to control memory consumption; concurrently, the buffered messages also need to be fetched from the disk streams and sent to target machines in batches to fully utilize the network bandwidth. As a result, an OMS cannot be implemented simply as an append-only file like S^E, and we device a new structure *splittable stream* to achieve this goal. We also design a ring-based sending strategy to ensure balanced network traffic, and memory-efficient approaches for message combining and receiving through external memory (EM) merge-sort.

We will show that despite the additional in-memory buffers required by disk streams and message transmission, our approach keeps the memory bound established by Lemma 1. We also remark that this approach is just a baseline, and our ID recoding technique to be presented in Section 5 further eliminates the need of maintaining the IMS S^I and performing any EM merge-sort.

3.3.1 Outgoing Message Streams

In our target setting, disk streaming bandwidth is much higher than network bandwidth. Therefore, newly-generated messages are appended to OMSs first for later fetching and sending, and as long as message appending and transmission are well overlapped, disk IO cost is hidden by the communication cost. We now describe the OMS structure and how buffered messages are sent.

OMS Structure. We implement an OMS as a *splittable stream* that supports concurrent data appending and fetching. Specifically, a splittable stream S breaks a long stream of data items into multiple files $F_1, F_2, \ldots, F_j$, and it appends data items to the last file F_j in a streaming manner. Given a **file size parameter** $\mathcal{B}$, S appends a data item o by checking whether F_j's size will be larger than $\mathcal{B}$ after appending o: (1) if so, F_j is closed and a new file F_{j+1} is created for appending o; (2) otherwise, o is directly appended to F_j. It is not difficult to see that each file either has size at most $\mathcal{B}$, or contains only one data item whose size is larger than $\mathcal{B}$. We shall discuss how to set $\mathcal{B}$ shortly.

Since S writes to only one file at any time in a streaming manner, S requires only $b = 64$ KB memory space. In GraphD, since every OMS is organized as a splittable stream, the $|\mathbb{W}|$ OMSs in a machine take $|\mathbb{W}| \cdot b$ bytes of memory in total. Even when $|\mathbb{W}| = 1000$, all OMSs take merely 64 MB of RAM.

Sending Messages in OMSs. When an OMS S_i^O is writing F_j, messages in $F_1, \ldots, F_{j-1}$ can be sent to machine W_i in parallel. As Figure 2 shows, in GraphD, each machine W_i maintains an in-memory sending buffer B_{send}. A fully-written file split F_k of an OMS S_j^O is sent to W_j by first loading messages in F_k to B_{send}, which are then sent to W_j in one batch. Obviously, the buffer size $|B_{send}|$ should be at least the largest possible size of a file split.

We shall discuss how to set $|B_{send}|$ shortly. Now, we describe how we set $\mathcal{B}$. Obviously, the smaller $\mathcal{B}$ is, the finer-grained each file split is, and thus the less likely that message sending will be stalled on a file that is being appended. However, since messages are sent in batches of size around $\mathcal{B}$, $\mathcal{B}$ cannot be too small as sending messages in small batches is inefficient. GraphD sets $\mathcal{B}$ as 8 MB by default, which is tested to strike a good balance between the two aspects mentioned above, and keeps file number tractable.

Sending Strategies. Referring to Figure 2 again, each machine W_i orders the $|\mathbb{W}|$ OMSs into a ring, where each OMS keeps track of the batch number of the last file that has been sent (resp. fully written), denoted by n_s (resp. n_w). For example, for S_j^O in Figure 2 which is currently appending messages to F_5, $n_s = 2$ and $n_w = 4$. Moreover, each machine keeps track of the position in the ring, denoted by pos, from whose OMS (i.e., S_{pos}^O) the previous message file is selected to be loaded to B_{send} for sending.

If message combiner is not used, we scan through the ring from position pos, until an OMS S_j^O is reached whose $n_s < n_w$ (i.e., there is at least one file to send). There are two possible cases.

• **Case 1:** if such an OMS S_j^O is found before the scan reaches position pos again, we load F_{n_s+1} to B_{send} for sending, and then update pos as j. For example, for S_j^O in Figure 2, we only send F_3. Then, the same scan operation is repeated starting from the updated position in the ring. Note that even if S_j^O has more than one file to send to W_j (e.g., F_4 in Figure 2), the next scan will pick a file from another OMS $S_{j'}^O$ $(j' \neq j)$ for sending to $W_{j'}$ (if such a j' exists), to avoid communication bottleneck on the receiver-side. For the same reason, different machines will initialize position pos with different values when a job begins.

• **Case 2:** if the scan reaches position pos again without finding a valid OMS, then no OMS has a file to send, and thus the scanning thread goes to sleep. The thread is awakened to repeat the scan whenever a new message file is written.

On the other hand, if message combiner is used, we adopt a different scanning strategy to maximize the effect of message combining: if the scan locates a valid OMS, all its message files from F_{n_s+1} to F_{n_w} are combined for sending in one batch. Specifically, the messages are first merge-sorted (i.e., grouped) by destination vertex ID; then, another pass over the sorted messages combines each group into one message and appends this message to B_{send} for sending. The strategy is effective, since (1) when all active vertices have called *compute*(.) in the current superstep, OMSs are finalized and our strategy essentially combines all remaining messages in each OMS, while (2) otherwise, message combining runs in parallel with vertex-centric computation, and thus does not increase the computation time.

Only combined messages are appended to B_{send}. GraphD sets $|B_{send}|$ as $\mathcal{B}$ by default, but since messages for combining may come from multiple files (size of each bounded by $\mathcal{B}$), $|B_{send}|$ may need to increase beyond $\mathcal{B}$. However, since there are at most one combined message for each vertex in the target machine, $|B_{send}|$ is upper bounded by $O(\max_{W \in \mathbb{W}} |V(W)|)$. Thus, if combiner is used, GraphD increases $|B_{send}|$ to

$O(\max_{W \in \mathbb{W}} |V(W)|)$ (if originally smaller). Note that $|B_{send}|$ keeps the $O(\frac{|V|}{|\mathbb{W}|})$ memory bound established by Lemma 1.

Also, merge-sorting message files consumes only a small constant amount of memory space. To see this, assume that we sort files $F_1, F_2, \ldots, F_n$ by k-way merge-sort, then it takes $\lceil \log_k n \rceil$ sequential passes over all the messages. At any time during the merge-sort, only one merge operation is running where (at most) k sorted message files are being merged into one larger message file. Since we treat each sorted message file as a stream when reading/appending messages, the merge-sort uses $(k+1)$ small in-memory buffers, which takes $(k+1)b$ memory space.

GraphD sets k to 1000, and thus a merge-sort operation takes merely $(64 \text{ MB} + 64 \text{ KB})$ memory space. Moreover, the large value of k allows merge-sort to take only one pass even for very large graphs, since the number of message files to combine is usually smaller than $k = 1000$. To see this, recall that each message file has size around $\mathcal{B} = 8$ MB, and thus k files have size around 8 GB, which is quite large for an OMS (which only contains messages transmitted between one pair of machines).

3.3.2 Incoming Message Stream

Since outgoing messages are loaded to B_{send} and sent in batches, each machine also needs to maintain an in-memory receiving buffer B_{recv} with $|B_{recv}| = |B_{send}|$. In each machine, a receiving thread listens on the network, and uses B_{recv} to receive one message batch at a time. All received message batches constitute the content of the IMS S^I for use by the next superstep.

In a superstep, each active vertex v calls *compute*(*msgs*), where *msgs* is obtained from S^I. Since the vertex-state array A and edge stream S^E are already ordered by vertex ID, we require messages in S^I also to be ordered by destination vertex ID, so that vertex-centric computation may simply proceed in one pass over A by sequentially reading from both S^I and S^E. Specifically, to call v.*compute*(*msg*), v may read the next $d(v)$ items from S^E to obtain $\Gamma(v)$, and sequentially read all messages targeted at v from S^I and append them to *msgs*. The sequential read ends when a message targeted at $u > v$ (or the end of S^I) is reached.

However, the order that messages in S^I are received depends on the actual communication process. We adopt the following approach to make S^I ordered: whenever a machine receives a batch of messages in B_{recv}, it sorts the messages by destination vertex ID, and then writes the sorted messages to a file on disk; when all incoming messages for the current superstep are received, the sorted message files are then merged into one sorted message file S^I by merge-sort. Like in message combining, merge-sort takes merely $(64 \text{ MB} + 64 \text{ KB})$ memory space. Moreover, since each received message batch has size around 8 MB, when there are no more than 8 GB messages, the message files are simply merged; merge-sort is unlikely to take more than 2 passes since this requires a machine to receive over 8 TB messages.

3.4 Cost Analysis & Other Issues

We now show that the total memory cost incurred by IMS and OMSs on each machine is a small constant and thus does not influence the $O(|V|/|\mathbb{W}|)$ memory bound established by Lemma 1, assuming a small cluster where $|\mathbb{W}| < 1000$. For communication, each machine maintains two buffers B_{send} and B_{recv} which take $2\mathcal{B} = 16$ MB memory space. For computation, appending messages to the OMSs needs $|\mathbb{W}| \cdot b < 64$ MB memory, and reading S^E and S^I needs $2b = 128$ KB memory. When combiner is used, the merge-sort for combining messages, and the merge-sort for constructing S^I, each takes $(64 \text{ MB} + 64 \text{ KB})$ memory. Therefore, each machine requires around 200 MB memory besides that for the vertex-state array A, well affordable by a modern PC.

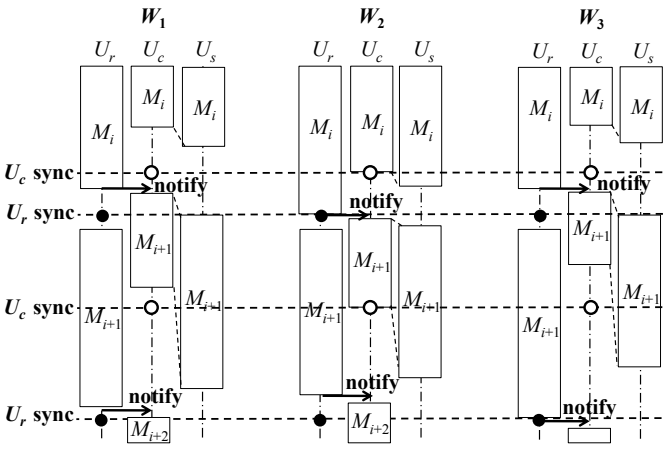


Fig. 3. An Illustration of the Parallel Framework

In each superstep, all the streams S^E, S^I and S_i^O are sequentially read and/or written for only one pass, while the merge-sort for combining messages (resp. for constructing S^I) takes one (or for a giant graph, two) additional pass over the outgoing (resp. incoming) messages. Thus, the disk IO cost is low.

Data loading from HDFS is similarly processed as message passing, except that data items in an OMS and an IMS are now vertices (along with their adjacency lists) rather than messages. This additionally requires that $|B_{send}|$ and $|B_{recv}|$ be at least large enough to hold the highest-degree vertex and its adjacency list during loading. The received vertices are merge-sorted by vertex ID into S^I, which is then split into A and S^E in one pass.

GraphD also supports algorithms that perform topology mutation. Edge mutations are performed in v.*compute*(.) by directly updating $\Gamma(v)$, which is written to a new local edge stream. Vertex mutations are performed after vertex-centric computation, where new vertices are appended to the vertex-state array A, and deleted vertices are simply masked in A. Our design of streams also naturally supports checkpointing, by periodically backing up current streams to HDFS for later recovery.

4 Parallel Framework of DSS

We now introduce how GraphD utilizes the components described in Section 3 for parallel graph computation, to overlap computation (disk streaming) with communication (message transmission).

Specifically, each machine runs three units in parallel: (1) a **sending unit** U_s that sends outgoing messages; (2) a **receiving unit** U_r that receives incoming messages; and (3) a **computing unit** U_c that performs vertex-centric computation (to generate messages). Parallelism within each machine is realized through the interaction of the three units as illustrated by Figure 3, which we shall explain in more detail in the following paragraphs. We first present our two (realistic) assumptions that our parallel framework lies on: (i) messages transmitted on a channel between a pair of machines are received in the same order as they were sent, which is guaranteed by TCP connections; (ii) we use *condition variables* to avoid a waiting thread from occupying CPU resources, which is implemented by std::condition_variable of C++ 11.

Synchronization Between Supersteps. GraphD implements Pregel's synchronous execution model. Since network bandwidth is the bottleneck, it is unreasonable to delay the transmission of messages generated in Step i, by transmitting messages generated in Step $(i+1)$. For example, consider Step i in Figure 3: machine W_3 finishes receiving messages (by U_r) earlier than W_2 and starts to compute Step $(i+1)$ (by U_c). If W_3 transmits the generated messages (by U_s) immediately, its messages towards W_2 will compete for the network bandwidth and delay W_2's progress for Step i since W_2 is still receiving messages generated in Step i.

Therefore, the sending unit U_s of every machine should block the sending of messages generated by its computing unit U_c in Step $(i+1)$, until all messages generated in Step i have been received by the receiving units U_r of all machines. GraphD guarantees this property, by letting U_r in each machine synchronize with the receiving units of all other machines, after it has received all the messages generated in Step i towards its machine (we will discuss how U_r determines this condition shortly). After the synchronization, U_r guarantees that all messages generated in Step i have been transmitted, and thus it notifies U_s (through a condition variable) to send messages generated in Step $(i+1)$.

Referring to Step i in Figure 3 again, U_r of all machines synchronize right after they finish receiving messages, as indicated by the first dashed line marked with "U_r sync" (2nd horizontal line). Now consider W_3: even though U_c starts computing Step $(i+1)$ before "U_r sync", U_s blocks until U_r passes "U_r sync".

Message Receiving. We now explain how U_r decides whether it has received all messages of Step i. Specifically, whenever U_s in a machine W_j has sent all its messages towards another machine W_k (i.e., W_j's OMS S_k^O is exhausted), it will send a special "end tag" to W_k. As a result, a machine W_k just needs to count the number of end tags received, and if it reaches $|\mathbb{W}|$, messages from all machines must have been received. This is correct because the previously mentioned "U_r sync" guarantees that all messages (including end tags) generated in Step i must be transmitted before any message (including an end tag) generated in Step $(i+1)$.

Here, U_s decides that it has exhausted its OMS S_k^O (and sends an end tag to W_k) if the following two conditions are both met: (1) U_c has finished vertex-centric computation for Step i, and will thus generate no more messages of Step i; and (2) there is no more message file in OMS S_k^O for sending.

Vertex-Centric Computation. To call v.*compute*(*msgs*) in Step $(i+1)$, we need to guarantee that *msgs* contains all the messages targeting v from Step i. Therefore, when U_c finishes its computation of Step i, it has to be blocked until U_r has received all messages towards it generated in Step i; then U_r notifies U_c to start computing Step $(i+1)$. Referring to W_3 in Figure 3 again, U_c finishes computing Step i much earlier than U_r finishes receiving messages; but U_c has to wait for U_r to get all necessary messages and notify it, before starting to compute Step $(i+1)$.

However, unlike U_s, U_c does not need to wait till all receiving units are synchronized, and may start generating messages of Step $(i+1)$ earlier, although these messages will only be sent by U_s after the synchronization. Referring to W_3 in Figure 3 again, U_c starts computing Step $(i+1)$ before "U_r sync".

To summarize, in Step i, U_r first keeps receiving messages until $|\mathbb{W}|$ end tags are received, then notifies U_c that it is allowed to compute Step $(i+1)$, then synchronizes with the receiving units of the other machines; and if the job should continue, U_r then notifies U_s that it is allowed to send messages for Step $(i+1)$.

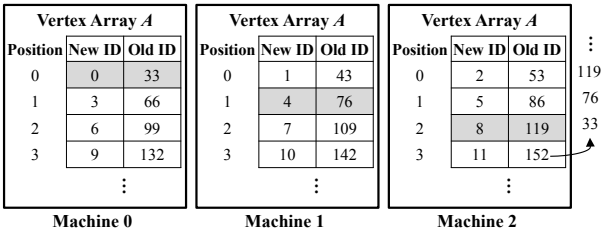


Fig. 4. Example of ID Recoding

The benefit of letting U_c start computing Step $(i+1)$ earlier is that, when U_s starts to send messages of Step $(i+1)$, it can readily find fully-written OMS files for sending, and thus network bandwidth can be fully utilized.

Synchronization of Global Information. When U_c of a machine W performs vertex-centric computation in Step i, it aggregates data to its local aggregator, and updates local control information such as whether W has sent any message and whether any vertex is active after calling *compute*(.). These data need to be synchronized to decide whether to continue computing Step $(i+1)$, and to obtain the global aggregator value for use by *compute*(.) in Step $(i+1)$. We let the computing units of all machines synchronize these global data as soon as they finish their vertex-centric computation, and there is no need to wait for the slower message transmission to complete. For example, in Figure 3, we can see that in each superstep, synchronization among U_c of all machines, indicated by dashed line marked with "U_c sync", is before "U_r sync". This allows U_c to start computing a new superstep much earlier than the synchronization among receiving units. For example, in Figure 3, W_3 starts computing Step $(i+1)$ (by U_c) before "U_r sync" of Step i. If U_c decides that the job should terminate after synchronizing with other computing units, it signals U_s and U_r to terminate after they finish processing their current superstep, and then terminates itself.

5 The ID-Recoding Technique

Many Pregel algorithms use message combiner to reduce communication workload. For these algorithms, GraphD supports a more efficient execution mode, which uses a technique called ID-recoding to (1) directly digest incoming messages in memory which eliminates S^I, and to (2) combine outgoing messages in memory which eliminates the need of external-memory merge-sort on OMS file splits, while (3) retaining the $O(\frac{|V|}{|\mathbb{W}|})$ memory bound established by Lemma 1. As a result, each superstep only requires one sequential pass over the edge stream S^E and over the generated messages (through message appending to OMSs). In contrast, Pregelix performs expensive external-memory sort and group-by operations even for algorithms where combiner applies.

Vertex ID Recoding. The key idea of ID recoding is to establish an efficient-to-compute one-to-one mapping between the ID of a vertex and its position in the state array A. Before introducing how GraphD establishes this mapping, we first present our underlying assumptions. In GraphD, machines are numbered by $0, 1, \cdots, |\mathbb{W}|-1$, and the vertex IDs are to be recoded into $0, 1, \cdots, |V|-1$. When GraphD runs in *recoded mode*, it uses the vertex partitioning function $hash(v) = id(v)$ modulo $|\mathbb{W}|$.

As an illustration, Figure 4 shows the vertex state arrays A in a cluster of 3 machines, where for each vertex, we show its old ID and new (i.e., recoded) ID. We can see that the old IDs

are sparsely numbered as 33, 43, 53, $\cdots$, and are recoded into dense new IDs 0, 1, 2, $\cdots$. In the recoded mode, new IDs are used as the actual vertex ID, and our recoding scheme guarantees that after recoding, the worker of a vertex v can still be computed by hashing v's (new) ID, i.e., $hash(v) = id(v)$ modulo $|\mathbb{W}|$. For example, for the second vertex in A of Machine 2, its new ID is 5, and 5 modulo 3 is equal to 2, which is exactly the machine ID.

For a vertex at position pos of array A in Machine i, we can compute its new ID as $(|\mathbb{W}| \cdot pos + i)$. For example, in Figure 4, the vertex whose old ID is 86 is at position 1 of array A in Machine 2, and thus its new ID is computed as $(3 \cdot 1 + 2) = 5$. Moreover, given the new ID of a vertex, id, on Machine i, we can compute its position in A as $\lfloor id/|\mathbb{W}| \rfloor$. For example, in Figure 4, the vertex whose new ID is 5 (in Machine 2) is at position $\lfloor 5/3 \rfloor = 1$.

Preprocessing. To run a job in recoded mode, we need to preprocess the graph to assign its vertices with new IDs $0, 1, \cdots, |V|-1$. We now describe our preprocessing algorithm, which is essentially a GraphD job running in the basic mode as presented in Section 3, and thus requires only $O(\frac{|V|}{|\mathbb{W}|})$ memory on each machine.

During preprocessing, old IDs are used as the vertex ID for vertex-to-machine assignment and vertex-to-vertex message passing. After the input graph is loaded, each machine W_i scans its array A and assigns each vertex (at each position pos) a new ID $(|\mathbb{W}| \cdot pos + i)$. However, for each vertex v, the neighbor IDs in $\Gamma(v)$ (which are stored in S^E) are still the old IDs. For example, in Figure 4, we show the adjacency list of the vertex at position 3 in array A of Machine 2, whose neighbors are vertices with old IDs 33, 76, 119, etc. (i.e., the gray vertices in Figure 4); even though the vertex is assigned a new ID 11, its adjacency list items in S^E are still the old IDs 33, 76, 119, etc. We need to replace them with their new IDs, 0, 4, 2, etc., since new IDs will be used for message passing in recoded mode.

Recoding the IDs in S^E (i.e., adjacency lists) takes 3 supersteps. Let us denote the old (resp. new) ID of a vertex v by $id_o(v)$ (resp. $id_n(v)$). In Step 1, each vertex v sends $id_o(v)$ to every out-neighbor $u \in \Gamma(v)$ asking for $id_n(u)$. For example, the vertex with $id_o(v) = 152$ in Figure 4 sends messages to neighbors 33, 76, 119, etc., asking for their new IDs 0, 4, 8, etc. In Step 2, a vertex u responds to each requester $id_o(v)$ by sending $id_n(u)$ to it. Continuing with the previous example, when vertices 33, 76 and 119 receive vertex 152's ID, they will send their new IDs 0, 4 and 8 to vertex 152, respectively. Finally, in Step 3, each vertex v simply appends the received new neighbor IDs to a new edge stream S^E_{rec}, which is the edge stream for use in recoded mode. Continuing with the previous example, vertex 152 simply appends the received new neighbor IDs 0, 4, 8, etc. to S^E_{rec}. Note that the whole recoding process sends only $O(|E|)$ messages.

For an undirected graph, we can skip Step 1 since a vertex u can directly send $id_n(u)$ to each neighbor $v \in \Gamma(u)$.

Execution in Recoded Mode. After a graph is recoded as mentioned above, state array A and stream S^E_{rec} of each machine are already on its local disk; our recoded mode thus simply lets each machine load A to memory, and stream S^E_{rec} on local disk.

Additionally, users are required to specify an **identity element** e^0, which when combined with any message m, gives the combined message whose value is still m. For example, $e^0 = 0$ for PageRank computation since $e^0 + m = m$; while if the combiner's operation is to take minimum rather than sum, e^0 can be set as ∞.

○ ***In-Memory Message Digesting.*** In recoded mode, U_r now directly digests messages in memory, eliminating the need of constructing S^I using EM merge-sort. To achieve this goal, in each step i before receiving messages, U_r first creates an in-memory array with $|V(W)|$ message elements, denoted by A_r. Here, $A_r[pos]$ refers to the combined message targeting the vertex at $A[pos]$. Each element in A_r is initialized as e^0. For example, for Machine 2 in Figure 4, U_r creates an array A_r where $A_r[1]$, $A_r[2]$, $A_r[3]$ and $A_r[4]$ corresponds to combined messages to be received by vertices 2, 5, 8 and 11, and if the job performs PageRank computation, all elements in A_r are initialized as 0.

When a batch of messages is received into B_{recv}, for each message, we compute the position of its destination vertex u in array A from u's ID, i.e., $pos = \lfloor id(u)/|\mathbb{W}| \rfloor$, and then combine the message to $A_r[pos]$. For example, if Machine 2 in Figure 4 receives a message with value 0.2 targeting vertex 5, 0.2 will be simply added to $A_r[1]$ since $pos = \lfloor 5/3 \rfloor = 1$.

After all messages generated in Step i are received and U_c starts processing Step $(i+1)$, the corresponding vertex of $A[pos]$ is regarded as having received messages only if $A_r[pos] \neq e^0$, in which case *compute*(*msgs*) is called on the vertex with *msgs* containing only the combined message $A_r[pos]$. Continuing with our previous example about PageRank computation, now $A_r[pos]$ equals the sum of messages received by the vertex at $A[pos]$; and $A_r[pos] = 0$ (i.e., e^0) implies that the vertex has no message.

Finally, when U_c finishes computing Step $(i+1)$, it frees A_r from memory as messages from Step i are no longer needed.

Let us define $A_r^{(i)}$ as the array A_r that is created by U_r for receiving messages generated in Step $(i-1)$ and then freed by U_c after it finishes computing Step i. Then, two arrays of A_r coexist in any superstep: in Step i, U_r creates $A_r^{(i+1)}$ and updates it with received messages (for use by U_c in Step $(i+1)$), while U_c obtains incoming messages from $A_r^{(i)}$ for computation. The two arrays require $O(|V(W)|)$ additional memory, which still keeps the $O(\frac{|V|}{|\mathbb{W}|})$ memory bound established by Lemma 1.

○ ***In-Memory Message Combining.*** Similarly, U_s always maintains an in-memory array with $max_{W \in \mathbb{W}}|V(W)|$ message elements, denoted by A_s, for combining outgoing messages. This does not breach the $O(\frac{|V|}{|\mathbb{W}|})$ memory bound of Lemma 1.

Each element of A_s is initialized as e^0. Recall that U_s combines and sends those messages from one OMS (i.e., towards one destination machine) at a time. To combine a set of messages towards a machine W_i, for each message that targets at a vertex u, U_s computes its position in array A of the destination machine W_i, i.e., $pos = \lfloor id(u)/|\mathbb{W}| \rfloor$, and then combines the message to $A_s[pos]$. For example, assume that vertices 5 and 8 in Machine 2 in Figure 4 both send message 0.2 to vertex 4 in Machine 1; since $pos = \lfloor 4/3 \rfloor = 1$, both 0.2 values are added to $A_s[1]$ giving 0.4.

After all messages in an OMS are combined to A_s, for each message element $A_s[pos] \neq e^0$, U_s attach the message value with the ID of its destined vertex, i.e., $|\mathbb{W}| \cdot pos + i$ (i is the destination machine ID); U_s then appends the target-labeled message to B_{send} for sending. Continuing with the previous example where $A_s[1] = 0.4$ in Machine 2, then U_s will label this combined message with the destined vertex ID 4, computed as $3 \cdot 1 + 1$.

To guarantee that all elements of A_s are e^0 before combining the next batch of message files, U_s sets each $A_s[pos]$ ($\neq e^0$) back to e^0 after the corresponding message gets appended to B_{send}.

○ ***Topology Mutation.*** Topology mutation is handled similarly as in the basic mode, with a change for vertex addition. Specifically, in a superstep, after vertex-centric computation, U_c first recodes the IDs of the newly added vertices by synchronizing with the

TABLE 1
Graph Datasets

Data	Type	$\|V\|$	$\|E\|$	AVG Deg	MAX Deg
Twitter	directed	52,579,682	1,963,263,821	37.34	779,958
WebUK		133,633,040	5,507,679,822	41.21	22,429
ClueWeb		978,408,098	42,574,107,469	43.51	7,447
Kron-32-16		2^{32}	67,971,861,142	15.83	14,454,242
Friendster	undirected	65,608,366	3,612,134,270	50.06	5,214
BTC		164,732,473	772,822,094	4.69	1,637,619

computing units of other machines, using the same method as preprocessing does; U_c then appends these recoded vertices to A. The overhead caused by the above intra-superstep id-recoding operation is proportional to the number of vertices added.

6 Experiments

This section reports the results of our empirical study of GraphD's performance, which is compared with other existing systems, under various hardware environments.

6.1 Experimental Setup

Datasets. Table 1 lists the graph datasets used for our evaluation. There are five real datasets: two directed web graphs *WebUK*[3] and *ClueWeb*[4]; two social networks *Twitter*[5] (directed) and *Friendster*[6] (undirected); and an RDF graph *BTC*[7]. To test system scalability, we also generated a giant synthetic graph *Kron-32-16* using Graph 500 Kronecker graph generator[8], where scale is 32 (i.e., there are 2^{32} vertices), and edge factor (i.e., $|E|/|V|$) is 16. We removed repetitive edges, and thus, the actual $|E|/|V|$ is 15.83.

Algorithms. We evaluate the performance of the systems using three well-studied Pregel algorithms: PageRank computation [15], single-source shortest path (SSSP) computation [15] and the Hash-Min algorithm of [31] for computing connected components.

PageRanks is only evaluated on directed graphs since it target vertices with directed links; Hash-Min is only evaluated on undirected graphs where connected component is defined; while SSSP is evaluated on both directed and undirected graphs.

Systems. The distributed out-of-core systems we compare against include Pregelix (Release 0.2.12), HaLoop and Chaos. The single-machine systems we compare against include GraphChi and X-Stream (v1.0). We also report the performance of representative in-memory systems, Pregel+ and Giraph (1.1.0), as a reference to measure the disk IO overhead incurred by out-of-core execution. The source code of GraphD and all the applications evaluated can be found at: http://www.cse.cuhk.edu.hk/systems/graphd.

6.2 Performance on a Cluster of PCs

Recall that GraphD target a small cluster of commodity PCs connected by Gigabit Ethernet. This set of experiments compare existing systems under this environment. We ran the systems using 16 desktop PCs in a lab classroom, each with four 3.40GHz cores (Intel Core i5-4670), 8GB RAM and a 320GB disk. The PCs are connected by an unmanaged switch, and we observed that the 1gbps network bandwidth cannot be able to fully reached.

3. http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05
4. http://law.di.unimi.it/webdata/clueweb12
5. http://konect.uni-koblenz.de/networks/twitter_mpi
6. http://snap.stanford.edu/data/com-Friendster.html
7. http://km.aifb.kit.edu/projects/btc-2009/
8. https://github.com/graph500/graph500/tree/master/generator

In our previous description of GraphD, we assumed that each machine runs only one process which consists of three threads for the units U_s, U_c and U_r. In order to better utilize the multi-core processors, we actually ran multiple processes on each machine. Here, each PC runs 2 processes (and thus streams 2 edge streams) concurrently. Running more processes does not help since the network and disk bandwidths are already saturated.

We do not include Chaos in this set of experiments, since Chaos is designed to run with high-speed network like 40GigE. As we shall see, in Section 6.7, the performance of Chaos is much poorer than other systems when GigE is used.

We denote the normal (resp. recoded) mode of GraphD by "GDBasic" (resp. "GDRecoded"). GDBasic, Pregel+, and Pregelix need to load graph data from HDFS; while GDRecoded only needs to let each machine load data from local disk. In contrast, HaLoop, GraphChi and X-Stream directly scan the disk-resident graph in each iteration, and there is no data loading phase.

Table 2 reports the running time of various systems on our PC cluster. For each system that loads data, we report the time as "loading time" + "computation time". For example, for Pagerank on *WekUK*, GDBasic takes 628.9s to load the graph, and 1189 seconds to run computation for 10 supersteps, while HaLoop takes 19954s for 10 supersteps and there is no loading phase.

Among the systems, GDRecoded needs to recode input graph first, and GraphChi needs to preprocess input graph into shards. There preprocessing times are reported in grey font in Table 2. For example, for Pagerank on *WekUK*, GDRecoded first recodes input graph by using 651.4s to load it, and 841.7s for the actual recoding, after which the recoded graph can be loaded from local disks in 1.74 seconds, and 10 supersteps of computation takes 982.3s; also, GraphChi needs to spend 2114s to shard *WekUK*, before the actual computation that takes 3614s.

For each experiment, we also mark the smallest "computation time" among all systems in red font labeled with a star.

PageRank. Table 2(a) reports the results of PageRank over three directed graphs. We only ran 10 iterations on *WebUK* and *Twitter* and 5 supersteps on *ClueWeb*, since each iteration takes roughly the same time, and each iteration is very time-consuming for all the other out-of-core systems that we compared with.

As Table 2(a) shows, Pregel+ can only process *Twitter* in our PC cluster due to the limited memory space, and it is even slightly slower than GDBasic and GDRecoded. This is because, network bandwidth is the bottleneck rather than disk IO, and GraphD's parallel framework fully hides the computation cost inside the communication cost; while in Pregel+'s implementation, message transmission starts after computation finishes (i.e., all messages are generated). On ClueWeb, GDRecoded takes only 4639s to finish 5 supersteps, much faster than GDBasic which takes 7920s; however, this is brought about due to graph recoding which takes 10956s, which is still worthwhile if we need to run PageRank for many computations till convergence. Also, the computation time of ID Recoding is consistently less than twice of the data loading time, and is thus an efficient preprocessing.

Among the other systems, Pregelix is much slower than GDBasic since it performs costly relational operations. X-Stream is generally much slower than GraphChi as also observed by [3]. HaLoop is sometimes slower than X-Stream (e.g., on *WebUK*) even though it uses all machines.

Hash-Min. Table 2(b) reports the results of Hash-Min over the two undirected graphs, where the number of supersteps is what it

TABLE 2
Performance on the PC Cluster (time unit: seconds; ⋆: smallest computation time among all systems)

	WebUK (10 supersteps)	ClueWeb (5 supersteps)	Twitter (10 supersteps)
GDBasic	628.9 + 1189	5835 + 7920	188.7 + 458.2
GDRecoded	651.4 + 841.7 + 1.74 + **982.3***	6020 + 10956 + 23.0 + **4639***	189.4 + 288.0 + 1.02 + **434.6***
Pregel+	Out of Memory	Out of Memory	187.7 + 480.6
Pregelix	426.3 + 7390	3221 + 13861	119.5 + 1419
HaLoop	19954	Out of Disk	3218
GraphChi	2114 + 3614	Out of Disk	622.2 + 1488
X-Stream	17669	Out of Disk	5989

(a) Performance of PageRank

	BTC (30 supersteps)	Friendster (22 supersteps)
GDBasic	116.8 + **81.7***	367.0 + 309.5
GDRecoded	112.4 + 51.3 + 1.25 + 82.4	380.5 + 273.3 + 1.08 + **279.9***
Pregel+	115.9 + 88.9	388.7 + 294.7
Pregelix	96.3 + 337.9	204.2 + 1397
HaLoop	8152 s	11534 s
GraphChi	217.3 + 353.4 s	1240 + 6815 s
X-Stream	2518 s	12012 s

(b) Performance of Hash-Min

	BTC (16 supersteps)	Friendster (23 supersteps)	WebUK (665 supersteps)	Twitter (16 supersteps)
GDBasic	170.3 + **1.70***	642.6 + 150.8	1152.8 + 191.6	335.2 + 69.1
GDRecoded	116.9 + 57.5 + 1.26 + 3.28	403.5 + 300.1 + 1.083 + **143.9***	667.1 + 914.2 + 2.86 + 223.8	199.5 + 286.0 + 1.04 + 65.8
Pregel+	177.1 + 2.24	Out of Memory	Out of Memory	334.0 + **54.1***
Pregelix	193.5 + 60.1	405.9 + 1648	620.0 + 24108	197.8 + 236.9
HaLoop	3729	10663	> 24 hr	3790
GraphChi	235.7 + 72.8	1150 + 10230	1884 + 41538	583.3 + 2017
X-Stream	1025	11803	> 24 hr	3102

(c) Performance of SSSP

takes to find all connected components. Similar to the PageRank experiments, GDBasic, GDRecoded and Pregel+ exhibit similar performance since network bandwidth is the bottleneck for them all, and Recoded even beats Pregel+ over *Friendster*.

The computation workload of Hash-Min is typically as follows: most vertices perform computation in the first few supersteps, but as computation proceeds, less and less vertices perform computation in a superstep, making the computation workload very sparse. Sparse workload is not a problem for in-memory systems since all adjacency lists are memory-resident; meanwhile, GraphD is also able to avoid accessing many useless adjacency lists with the help of its streaming function *skip*(*num_items*) which we introduced in Section 3.2. However, the other out-of-core systems do not have effective support for sparse workload, and thus as Table 2(b) show, their computation times are much longer than GraphD and Pregel+.

SSSP. Table 2(c) reports the results of SSSP over two directed graphs *WebUK* and *Twitter*, and two undirected graphs *BTC* and *Friendster*. The number of supersteps required are also shown. All edges were given weight 1, and thus the computation is essentially breadth-first search (BFS).

Unlike PageRank and Hash-Min, the computation workload of every superstep for BFS (or more generally, SSSP) is sparse. This is because in BFS, a vertex will only send messages to its neighbors when it is reached from the source vertex for the first time. Since every vertex sends messages along adjacent edges for only once during the whole period of computation, the total workload is merely $O(|E|)$, which amounts to the workload of just one superstep in PageRank computation.

Table 2(c) shows that Pregel+ beats all the out-of-core systems on *Twitter*, which is not surprising since Pregel+ keeps all adjacency lists in memory. GraphD is also comparable, thanks to the use of streaming function *skip*(*num_items*).

Surprisingly, on *BTC* and *WebUK*, GDBasic even outperforms GDRecoded. This is because, if there are too few messages to send in each superstep, the overhead of manipulating the additional arrays (i.e., A_r and A_s mentioned in Section 5) in recoded mode backfires. Note that all computations on *BTC* finished in seconds for both modes of GraphD, whose workload is really low. While computations on *WebUK* took a longer time, this is mainly because of the large number of supersteps (i.e., 665). After all, IO-Recode needs to create, update and tear down those large additional arrays for 665 times.

Also surprisingly, on *WebUK*, Pregelix is over two orders of magnitude slower than GraphD. We found that Pregelix incurs a fixed cost of at least 35 seconds for each superstep, while a superstep of GDBasic can be as low as 0.02–0.03 seconds.

Table 2(c) also shows that X-Stream is impractical for jobs that run many iterations of sparse-workload vertex computation, since it needs to stream all edges in each iteration. For example, X-Stream could not finish on *WebUK* after a whole day. In fact, the authors of X-Stream themselves admitted this problem at the end of Section 5.3 in [21].

Finally, graph loading in ID Recoding is faster than GDBasic. This is because during ID Recoding, S^E does not include edge weights. We only attach edge weights when we append recoded adjacency list items to S^E_{rec}.

6.3 Performance on a Cluster of High-End Servers

From now on, we evaluate the scalability of various systems on a more scalable cluster of 15 high-end servers, each with twelve 2.0GHz cores (two Intel Xeon E5-2620 CPUs), 48GB RAM and a 200GB disk. These servers are connected by Cisco C2960 switch which we observe to better utilize the network bandwidth than the unmanaged switch in our PC cluster. To better utilize the multi-core processors, we ran 8 GraphD processes on on each machine. Running more processes does not help since the network and disk bandwidths are already saturated.

The cluster additionally has access to a 2TB disk, allowing us to run single-machine systems over big graphs like *ClueWeb* and *Kron-32-16*, whose size exceeds the disk capacity of each server (e.g., the input file of *ClueWeb* has size exceeds 400GB).

In addition to Pregel+, we also include Giraph as a reference in-memory system. Although Giraph loads data, we only report the total time since Giraph reports times labeled “Initialize”, “Input Superstep”, “Setup”, etc., in addition to the time for running each superstep. Also, we only run Giraph in-memory mode, since we find that for graphs where in-memory mode runs out of memory, out-of-core Giraph still runs out of memory (also observed in [1]).

Table 3 reports the running time of various systems. Our observations are similar to those from Table 2, with some differences.

Firstly, the performance improvement of GDRecoded over GDBasic is much more significant than in the PC cluster. For example, while GDRecoded only reduces the time of PageRank computation over *ClueWeb* from 7920s (of GDBasic) to 4639s (less than 2x) in Table 2, it reduces the time from 7422s to 1003s (over 7x) in Table 3. This make ID recoding more favorable if PageRank computation is going to run for many iterations. The much better performance of GDRecoded is contributed by its elimination of EM merge-sort, whose cost cannot be fully hidden since the network bandwidth is better utilized now.

Also, by comparing Table 2 and Table 3, we can see that computation on the server cluster is also much faster than on the PC cluster, thanks to the better network bandwidth utilization of the server cluster.

TABLE 3
Performance on the High-End Cluster (time unit: seconds; ⋆: smallest computation time among all systems)

	WebUK (10 supersteps)	ClueWeb (5 supersteps)	Twitter (10 supersteps)	Kron-32-16 (5 supersteps)
GDBasic	141.5 + 1093	1271 + 7422	44.7 + 424.6	1839 + 8744
GDRecoded	157.8 +213.8 + 3.92 + 331.6	1786 + 3306 + 22.1 + **1003***	66.4 + 86.2 + 2.71 + **121.2***	2367 + 4233+ 32.9 + **1636***
Pregel+	70.3 + **234.9***	Out of Memory	28.5 + 135.7	Out of Memory
Giraph	Out of Memory	Out of Memory	1366	Out of Memory
Pregelix	144.0 + 1744	1847 + 13820	59.2 + 877.6	2304 + 16045
HaLoop	17532	Out of Disk	3607	Out of Disk
GraphChi	2048 + 1768	26604 + 15966	729.4 + 1166	32768 + 19563
X-Stream	11198	75637	4542	96381

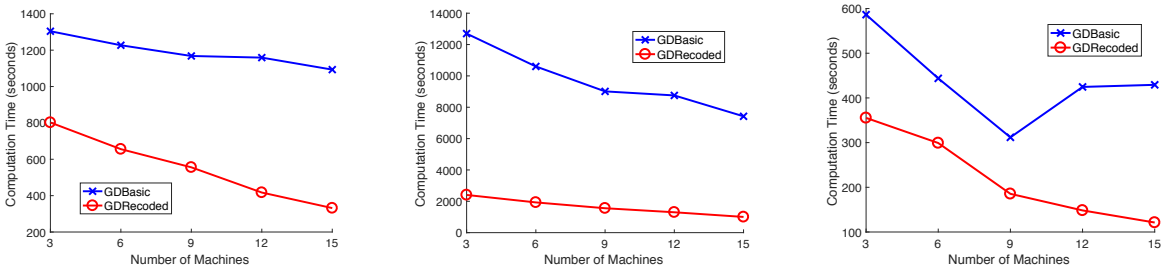
(a) Performance of PageRank

	BTC (30 supersteps)	Friendster (22 supersteps)
GDBasic	30.4 + 59.2	75.2 + 197.3
GDRecoded	33.5 + 14.3 + 2.44 + 34.5	75.6 + 96.9 + 2.58 + **94.8***
Pregel+	18.6 + **20.7***	47.2 + 104.8
Giraph	566.3	757.7
Pregelix	102.5 + 503.3	95.6 + 1236
HaLoop	9507	5817
GraphChi	169.2 + 550.0	1430 + 1808
X-Stream	124.8	6513

(b) Performance of Hash-Min

	BTC (16 supersteps)	Friendster (23 supersteps)	WebUK (665 supersteps)	Twitter (16 supersteps)
GDBasic	31.6 + 4.75	135.0 + 118.3	252.8 + 166.2	88.9 + 35.0
GDRecoded	30.7 + 29.0 + 2.57 + 9.06	55.1 + 78.8 + 2.23 + 66.2	134.3 + 202.2 + 2.65 + 253.6	65.7 + 72.3 + 2.33 + 25.3
Pregel+	25.3 + **1.59***	67.9 + **43.4***	102.7 + **74.4***	39.3 + **19.8***
Giraph	277.3	429.4	3270	232.1
Pregelix	172.1 + 200.5	137.3 + 568.3	186.6 + 3586	119.1 + 462.5
HaLoop	9016	3781	> 24 hr	1828
GraphChi	161.6 + 155.9	1478 + 2041	1922 + 14740	631.2 + 637.4
X-Stream	105.5	4943	> 24 hr	2413

(c) Performance of SSSP



(a) Machine Scalability on *WebUK* (b) Machine Scalability on *ClueWeb* (c) Machine Scalability on *Twitter*

Fig. 5. Scalability Results to Number of Machines

Among the totally 10 experiments shown in Table 3, Pregel+ beats GraphD in 6 of them. This is because GraphD's parallel execution framework is not able to fully hide the disk streaming overhead when network bandwidth utilization is high. However, GDRecoded still wins 4 experiments, and is close to Pregel+ in performance for the other experiments, which demonstrates the effectiveness of our parallel execution framework and ID recoding technique. Note that GDRecoded also wins 5 out of the 9 experiments in Table 2.

6.4 Scalability to the Number of Machines

While distributed computation allows each machine to only process a portion of the whole graph, this comes at the cost of network communication, which can form the performance bottleneck. The communication cost actually increases with the number of machines, since each machine has $|\mathbb{W}|$ message streams and thus the total number of message streams is $|\mathbb{W}|^2$, and the transmission of these streams contend for sender-side and receiver-side network bandwidth. When more machines than necessary is used, the increased communication cost may outweigth the benefit brought by workload dividing. This is also the reason why GraphD targets a small cluster.

We now demonstrate that GraphD scales out in a small cluster, by running PageRank computation over *WebUK*, *ClueWeb* and *Twitter* in our server cluster, with 3, 6, 9, 12, 15 machines, respectively, where each machine runs 8 processes. The performance results are reported in Figure 5. We can see that the performance of both GDBasic and GDRecoded improves as the number of machines increases, but the trend slows down as the number of machines become larger. In fact, Figure 5(c) shows that GDBasic performs the best when there are 9 machines, and the performance becomes poorer if we further increase machine number. This is because *Twitter* is relatively small, and with 9 machines, GDBasic already allows each process to process an affordable workload, but the increased communication cost begins to backfire.

TABLE 4
Time of Message Generation v.s. Message Transmission

		WebUK (10 steps)		ClueWeb (5 steps)		Twitter (10 steps)	
		M-Send	**M-Gene**	**M-Send**	**M-Gene**	**M-Send**	**M-Gene**
PC Cluster	**GDBasic**	1189 s	274.2 s	7920 s	4853 s	458.2 s	61.9 s
	GDRecoded	982.3 s	242.1 s	4639 s	2605 s	434.6 s	45.0 s
Server Cluster	**GDBasic**	1093 s	91.2 s	7422 s	2954 s	424.6 s	32.8 s
	GDRecoded	331.6 s	101.3 s	1003 s	613 s	121.2 s	35.7 s

In general, GDRecoded is much faster than GDBasic, and *WebUK* and *ClueWeb* are big enough so that increasing machine number all the way to 15 still keeps improving the performance of GDBasic and GDRecoded (as workload dividing is still very effective).

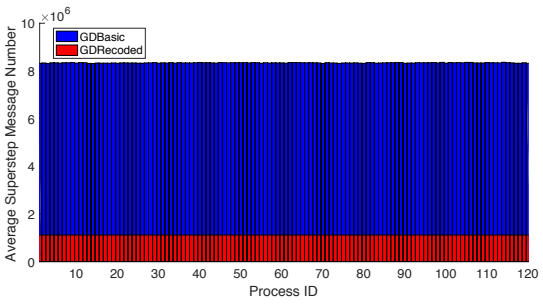
6.5 Cost of Communication v.s. Computation

We now demonstrate that network communication is the performance bottleneck of GraphD in both our PC cluster and our server cluster, by considering PageRank computation over *WebUK*, *ClueWeb* and *Twitter*.

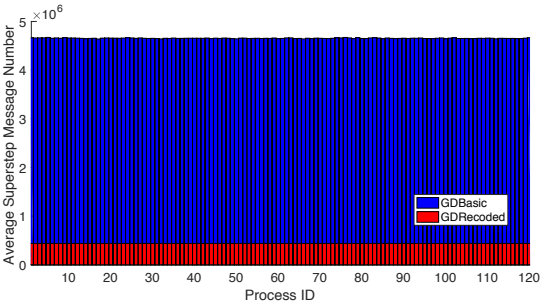
Recall that in each superstep, vertex-centric computation generates messages (by U_c), which get sent in parallel by U_s.

Table 4 shows the time taken by both GDBasic and GDRecoded to transmit messages (Column "M-Send"), and the time to generate messages (Column "M-Gene"). Since the behavior of U_c and U_s of different processes may vary due to imbalanced workload distribution (e.g., caused by vertex degree difference), we only report the time for the first process. All reported times are summed over all the 10 (or 5) supersteps, and "M-Gene" only sums the portion of time for vertex-centric computation.

We can see from Table 4 that in all the 6 data-cluster combinations, message transmission happens during the whole period of each superstep, but U_c only computes in the early stage (often much less than half) of each superstep. This demonstrates that network bandwidth is really the performance bottleneck.

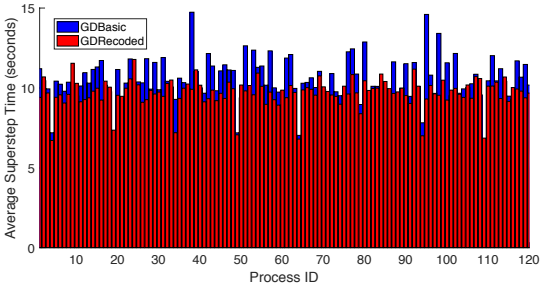


(a) Per-Superstep Message Number Distribution on *WebUK*

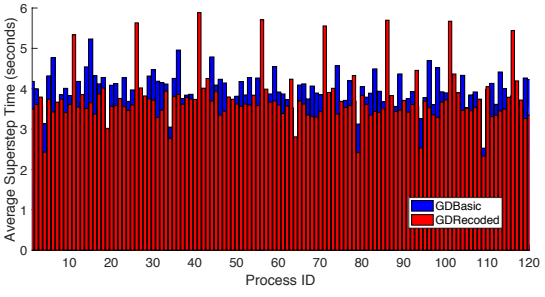


(b) Per-Superstep Message Number Distribution on *Twitter*

Fig. 6. Number of Messages Sent by Each Process in Each Superstep, Averaged over Five Supersteps of PageRank Computation



(a) Per-Superstep Computation Time Distribution on *WebUK*



(b) Per-Superstep Computation Time Distribution on *Twitter*

Fig. 7. Number of Messages Sent by Each Process in Each Superstep, Averaged over Five Supersteps of PageRank Computation

6.6 Workload Distribution

Recall from Section 3.2 that GraphD (normal mode) distributes vertices to processes using vertex ID hashing, which is exactly like in Pregel. However, vertex-based partitioning does not take vertex degree difference into consideration. For example, in Table 1, the average degree of *Twitter* (resp. *WebUK*) is 37.34 (resp. 41.21), but the maximum vertex degree is 779,958 (resp. 22,429). A high-degree vertex adds much more workload to its assigned process than an average vertex, and thus, we would like to check whether the workload of every processes are still reasonably balanced.

For both *WebUK* and *Twitter* (with uneven vertex degree distribution), we ran 5 supersteps of PageRank computation over GDBasic and GDRecoded (15 machines $\times$ 8 processes/machine), and count the average number of messages sent by each process in a superstep as shown in Figure 6, and the average time spent by each process in computing a superstep as shown in Figure 7.

Figure 7 shows that although the computation time varies a bit among the workers due to the dynamics of computation and communication, the distribution is still relatively balanced. Figure 6 shows that message workload distribution is highly balanced for both GDBasic and GDRecoded. This is because the large quantity of vertices on each process is able to average out the variance caused by the high-degree vertices assigned to this process. However, if one is willing to partitioning a big graph (which is often costly), it is straightforward to apply the two methods mentioned in [29] to GraphD: (i) partition-based ID recoding, or (ii) expanding vertex ID with process ID.

The most popular partitioning algorithm is Metis [9], as adopted by Mizan [10] and Giraph++ [26], which minimizes cross-partition edges (and hence minimize communication) in addition to balancing workloads. Integrating graph partitioning into GraphD may further improve its load balancing and the ultimate performance. GPS [22] and Mizan [10] further support dynamic vertex migration during computation to balance workloads, but the effectiveness may be limited as Section 3.4 of [28] reveals.

TABLE 5
Performance of Distributed Systems (Comparison with Chaos)

	Dataset	Pregel+	GDBasic	GDRecoded	Pregelix	Chaos
PageRank	**WebUK**	305.2 s	1235 s	335.5 s	1888 s	23651 s
	Twitter	164.2 s	469.3 s	123.9 s	936.8 s	4701 s
HashMin	**BTC**	39.3 s	89.6 s	36.9 s	605.8 s	11714 s
	Friendster	152.0 s	272.5 s	97.4 s	1332 s	17762 s
SSSP	**BTC**	26.9 s	36.4 s	11.6 s	372.6 s	3525 s
	Friendster	111.3 s	253.3 s	68.4 s	705.6 s	13509 s
	WebUK	177.1 s	419.0 s	256.3 s	3773 s	> 24 hr
	Twitter	59.1 s	123.9 s	27.6 s	581.6 s	3672 s

6.7 Comparison with Chaos

While Chaos [20] is also a distributed out-of-core graph engine, we have not included it in our comparison as it is designed to run with high-speed network, and [20] admits that the performance is undesirable with Gigabit Ethernet as we shall demonstrate here.

Unlike other distributed systems, Chaos does not support parallel data loading from HDFS. A graph needs to be converted into the binary format required by Chaos, and be distributed to machines before the actual computation. Without counting the preprocessing cost, Chaos is still much more expensive for graph computation than the other distributed systems we compared, mainly because it is designed only to run with a high-speed network. To demonstrate it, we run Chaos[9] in our server cluster to repeat our experiments. Table 5 shows the execution time of the distributed Pregel-like systems along with Chaos (at the last column). The execution time of Chaos reported in Table 5 does not include that for preprocessing, and we are not able to report results for the big datasets *ClueWeb* and *Kron-32-16*, since they cannot be preprocessed due to insufficient disk space. We can see from Table 5 that Chaos is significantly more expensive than the other distributed systems, which verifies that Chaos is not a good choice if high speed network is not available.

9. https://github.com/epfl-labos/chaos

TABLE 6
Space Cost of PageRank Computation over Twitter

	Memory	Disk
GDBasic	38.6 GB	32.3 GB
ID Recoding	37.9 GB	31.3 GB
GDRecoded	32.9 GB	36.1 GB
Pregel+	109.4 GB	—
Giraph	264.2 GB	—

	Memory	Disk
Pregelix	315.5 GB	66.7 GB
HaLoop	305.2 GB	96.9 GB
GraphChi	11.7 GB	48.1 GB
X-Stream	40.1 GB	29.9 GB
Chaos	114.3 GB	448.2 GB

6.8 Space Costs

Besides execution time, space usage is also an important metric of system scalability. Table 6 shows the peak usage of memory space and disk space, summed over all machines, when running PageRank computation over *Twitter*. We can see that GraphChi uses the least memory space, followed by our GraphD jobs, and then X-Stream. Pregelix executes with a dataflow engine that fully utilizes the available memory to reduce external-memory cost. Interestingly, Pregelix and HaLoop use even more aggregate memory space than in-memory systems Pregel+ and Giraph, possibly due to the space-consuming auxiliary structures for communication and for B-tree based vertex storage. Finally, we can see that on-disk data organization is much less compact in Chaos than in the other systems (over 10x w.r.t. GraphD), which explains why Chaos has poor performance in Table 5, i.e., the data request model of Chaos requires transmitting the huge amount of incompact data, which is very slow with Gigabit Ethernet.

6.9 Performance for Machine Learning

Pregel-like systems can also be used for machine learning algorithms that perform iterative computation, such as k-means clustering [19] and gradient descent. We implemented k-means clustering in GraphD to explore its performance and scalability. In this program, each vertex (i.e., data point) v maintains a value indicating which of the k centers is closet to v, while each coordinate of v is treated as an edge so that coordinates are streamed from S^E during computation. In a superstep, a vertex recomputes its closet center from the k centers aggregated from the last superstep, and then aggregates its coordinates to its new center in the aggregator which then computes new centers for the next superstep. The algorithm was implemented in normal mode, since vertices need not send messages and thus recoded mode does not help. We generated synthetic k-dimensional points using k Gaussian distributions with standard deviation 0.5, where the i-th Gaussian distribution is centered at the point whose i-th coordinate is 1 and whose other coordinates are 0. Points were generated from the Gaussian distributions in a round-robin manner, and the experiments were run on our high-end cluster.

The performance on GraphD is very good. For example, when there are 0.1 billion points of 5 (resp. 6) dimensions, each superstep takes 0.17s (resp. 0.19s); when there are 1 billion points of 5 (resp. 6) dimensions, each superstep takes 0.84s (resp. 1.15s).

6.10 Comparison with FlashGraph

We also tested the state-of-the-art single-machine SSD-based system FlashGraph, with a standalone server with 8GB RAM and a 128GB SSD (Samsung PM851 Series). We report the results of PageRank computation over *Twitter*.

After putting *Twitter* to FlashGraph's SAFS (a file system for SSD), the space used is 31.06 GB, comparable to GraphD as shown in Table 6. FlashGraph runs an asynchronous PageRank algorithm where only unconverged vertices perform computation, and thus the time of an iteration becomes shorter and shorter. It took 2763s and 31 iterations to finish PageRank computation on *Twitter* but the first 10 iterations are all over 120s (the first one takes 165s). In contrast, Table 2 shows that GraphD used less than 500s to finish 10 supersteps of PageRank computation on *Twitter*.

However, we remark that FlashGraph is a promising solution when an array of SSDs are available, since it uses only one machine and can work with multiple SSDs (while the server used in this experiment has only one SSD).

7 Conclusions

We presented a Pregel-like system, called GraphD, for efficient out-of-core processing of very large graphs with average computing resources that are readily available to most users. To process a graph $G = (V, E)$ with n machines using GraphD, we proved that each machine only requires $O(|V|/n)$ memory space. GraphD is also carefully designed to support sparse computation workload efficiently, to parallelize computation with communication, and to eliminate the need of any expensive external-memory operation by ID recoding. Open-source implementation of GraphD is provided, and extensive experiments demonstrated that GraphD's performance is competitive even when it is compared with an in-memory Pregel-like system.

While GraphD achieves impressive performance on a commodity cluster with a low bandwidth network, we remark that GraphD can be inferior to the state-of-the-art systems in other settings. For example, when the cluster memory is not a concern and network bandwidth is large enough, distributed in-memory systems such as GraM [27] can outperform GraphD as GraphD's design does not take advantage of the sufficiency in memory and network bandwidth to boost its performance. GraphD's performance can also be inferior to single-machine systems such as Ligra [24], Galois [17], and FlashGraph [32], which achieve superb performance with a many-core machine with big RAM (e.g., 1TB) or big flash memory. More detailed discussion on the different types of existing systems is given in Section 2.

Acknowledgments. This work was partially supported by Grants (CUHK 14206715 & 14222816) from the Hong Kong RGC, MSRA grant 6904224, ITF 6904079, and Grant 3132821 funded by the Research Committee of CUHK.

References

[1] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[3] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. VENUS: vertex-centric streamlined graph computation on a single PC. In *ICDE*, pages 1131–1142, 2015.

[4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[5] J. Gao, C. Zhou, J. Zhou, and J. X. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*, pages 556–567, 2014.

[6] 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.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

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

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

[10] 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.

[11] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *SC*, pages 830–841, 2016.

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

[13] 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.

[14] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3), 2015.

[15] 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*, pages 135–146, 2010.

[16] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, 2015.

[17] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[18] R. A. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC*, pages 1–11, 2010.

[19] 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.

[20] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, 2015.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

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

[23] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, pages 625–636, 2014.

[24] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory.

[25] G. M. Slota, S. Rajamanickam, and K. Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *IPDPS*, pages 293–302, 2016.

[26] 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.

[27] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

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

[29] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[30] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[31] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.

[32] D. Zheng, D. Mhembere, R. C. Burns, J. T. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

[33] C. Zhou, J. Gao, B. Sun, and J. X. Yu. Mocgraph: Scalable distributed graph processing using message online computing. *PVLDB*, 8(4):377–388, 2014.

Da Yan is an assistant professor with the Department of Computer Science at the University of Alabama at Birmingham. He is the winner of the 2015 Hong Kong Young Scientist Award, and he also led the BigGraph@CUHK project (see http://www.cse.cuhk.edu.hk/systems/graph). His research interests include Big Data analytics, distributed systems, graph data management, geospatial data management, and data mining.

Yuzhen Huang received his Bachelor degree in Computer Science from Sun Yat-sen University. He is currently a PhD student in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include analysis in massive temporal graphs and distributed systems.

Miao Liu received his Bachelor degree in Software Engineering from Shenzhen University. He is currently a research assistant in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include large-scale graph mining and distributed systems.

Hongzhi Chen is an MPhil student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. He is interested in distributed computing systems and large-scale graph processing.

James Cheng is an assistant professor with the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research focuses on big data infrastructures, distributed computing systems, and large-scale network analysis.

Huanhuan Wu received his Bachelor degree in Computer Science and Technology from Zhejiang University. He is currently a PhD candidate in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include analysis in massive temporal graphs and non-trivial distributed algorithms.

Chengcui Zhang is a Professor of Computer Science at the University of Alabama at Birmingham (UAB). She works in the broad areas of multimedia databases and information retrieval, multimedia data mining, multimedia security and forensics, Geoinformatics, and applied Bioinformatics. She has published over 150 refereed articles, many at the top tier venues in computer sciences including IEEE Transactions, IEEE Multimedia, ACM Multimedia (MM), IEEE Intl. Conf. on Data Mining (ICDM), ACM Conf. on Communication and Computer Security (CCS), and IEEE Intl. Conf. on Multimedia and Expo (ICME). Dr. Zhang's research has been externally supported by NSF, NIH, and by awards/gifts from the industry, including IBM, eBay, and Comcast. Dr. Zhang was the former Chair of IEEE Technical Committee on Semantic Computing and has served as the Conference Program Chair for many IEEE Conferences. She is an Associate Editor of IEEE Trans. on Multimedia and Intl. J. of Multimedia Data Engineering and Management.