VENUS: A System for Streamlined Graph Computation on a Single PC

Qin Liu, Jiefeng Cheng, Zhenguo Li, and John C.S. Lui, *Fellow, IEEE*

Abstract—Recent studies show that disk-based graph computation systems on just a single PC can be as highly competitive as cluster-based systems on large-scale problems. Inspired by this remarkable progress, we develop VENUS, a disk-based graph computation system which is able to handle billion-scale graphs efficiently on a commodity PC. VENUS adopts a novel computing architecture that features vertex-centric "streamlined" processing – the graph is sequentially loaded and an update function is executed for each vertex in parallel on the fly. VENUS deliberately avoids loading batch edge data by separating read-only structure data from mutable vertex data on disk, and minimizes random IOs by caching vertex data in the main memory whenever possible. The streamlined processing is realized with efficient sequential scan over massive structure data and fast feeding the update function for a large number of vertices. Extensive evaluation on large real-world and synthetic graphs has demonstrated the efficiency of VENUS. For example, to run the PageRank algorithm on a Twitter graph of 42 million vertices and 1.4 billion edges, Spark needs 8.1 minutes with 50 machines and GraphChi spends 13 minutes using high-speed SSD, while VENUS only takes 5 minutes on one machine with an ordinary hard disk.

Index Terms—graph computation, disk-based computing, vertex-centric streamlined processing.

1 Introduction

We are living in a "big data" era due to the dramatic advance made in the ability to collect and generate data from various sensors, devices, and the Internet. Consider the Internet data. The web pages indexed by Google were around one million in 1998, but quickly reached one billion in 2000 and have already exceeded one trillion in 2008. Facebook also achieved one billion users in 2012. It is of great interest to process, analyze, store, and understand these big datasets, in order to extract business value and derive new business model. However, researchers are facing significant challenges in managing these big datasets with our current methodologies and data mining software tools.

Graph computing over distributed or multi-core platform has emerged as a new framework for big data analytics, and it draws intensive interests recently [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. Notable systems include Pregel [1], GraphLab [2], and GraphChi [3]. They use a vertex-centric computing model, in which the user provides a simple update function to the system which is executed for each vertex in parallel [1], [2], [3]. These developments substantially advance our ability to analyze large-scale graph data that cannot be efficiently handled by previous parallel abstractions such as MapReduce [12] due to the sparse computation dependencies and iterative operations common in graph computation [3].

Distributed computing systems such as Spark [13], Pregel [1], PEGASUS [5], and GraphLab [2] can handle billion-scale graphs, but the cost of having and managing a large cluster is prohibitory for most users. On the other hand, disk-based single machine graph computing systems such as GraphChi [3], X-Stream [14], and TurboGraph [15] have shown great potential in big graph analytics. For example, to run the belief propagation algorithm on a Web graph with 6.7 billion edges, PEGASUS takes 22 minutes with 100 machines [5], while GraphChi uses 27 minutes on a single PC [3]. This striking result suggests that disk-based graph computation on a single PC is not only highly competitive even compared to parallel processing over large clusters, but it is very affordable also.

In general, graph computation is performed by iteratively executing the update function for each of vertices. The disk-based approach organizes the graph data into a number of *shards* on disk, so that each shard can fit in the main memory. Each shard contains all needed information for computing updates for a number of vertices. One iteration will execute all shards. A central issue is how to manage the computing states of all shards to guarantee the correctness of processing, which includes loading graph data from disk to the main memory, and synchronizing intermediate results to disk so that the latest updates are visible to subsequent computation. Therefore, there is a huge amount of data to be accessed per iteration, which can result in extensive IOs and becomes a bottleneck of the disk-based approach. This generates great interests in developing new architectures for efficient disk-based graph computation.

The seminal work for disk-based graph computation is the GraphChi system [3]. It organizes a graph into shards and processes each in turn. To execute a shard, the entire shard – its vertices and all of their incoming and outgoing edges – must be loaded into memory before processing. This constraint hinders the parallelism of computation and IO. In addition, after the execution, the updated vertex values need to be propagated to all the other shards in disk, which results in extensive IOs. The X-Stream system [14] explores a different, edge-centric processing (ECP) model. However, it is done by writing the partial, intermediate results to disk for subsequent processing, which doubles the sequential IOs while incurring additional computation cost and

- *Qin Liu and John C.S. Lui are with the Department of Computer Science and Engineering, the Chinese University of Hong Kong. E-mail: {qliu,cslui}@cse.cuhk.edu.hk*
- *Jiefeng Cheng and Zhenguo Li are with Huawei Noah's Ark Lab. E-mail: {cheng.jiefeng,li.zhenguo}@huawei.com The corresponding author of this paper is Jiefeng Cheng.*

data loading overhead. Since the ECP model uses very different APIs from previous vertex-centric graph computation systems, the user needs to re-implement many graph algorithms on ECP which causes high development overhead. Moreover, certain important graph algorithms such as community detection [16] cannot be implemented on the ECP model (explained in Section 5.2).

In this work, we present VENUS, a disk-based graph computation system that is able to handle billion-scale problems very efficiently on a moderate PC. Our main contributions are summarized as follows.

A novel computing model. VENUS supports the vertex-centric computing model with *streamlined processing*. We propose a novel graph storage scheme which allows to *stream* in the graph data while performing computation. The streamlined processing can exploit the large sequential bandwidth of a disk and parallelize computation and disk IO. Particularly, the vertex values are cached in a buffer in order to minimize random IOs, which is much more desirable in disk-based graph computation where the cost of disk IO is often a bottleneck. Our system also significantly reduces the amount of data to be accessed, generates much fewer shards than the existing scheme [3], and effectively utilizes the main memory with a provable performance guarantee.

Two new IO-friendly algorithms. We propose two IO-friendly algorithms to support efficient streamlined processing. In managing the computing states of all shards, the first algorithm stores vertex values of each shard into the corresponding files for fast retrieval during the execution. It is necessary to update on all such files timely once the execution of each shard is finished. The second algorithm applies merge-join to construct all vertex values on the fly. Our two algorithms adapt to memory scaling with less sharding overhead, and smoothly turn into the in-memory mode when the main memory can hold all vertex values.

A new analysis method. We analyze the performance of our vertex-centric streamlined processing computing model and other models, by measuring the amount of data transferred between disk and the main memory per iteration. We show that VENUS reads (writes) significantly less amount of data from (to) disk than other existing models including GraphChi. Based on this measurement, we further find that the performance of VENUS improves gradually as the memory increases, until an in-memory model is emerged where the least overhead is achieved; in contrast, existing approaches cannot unify the in-memory model and the disk-based model in a natural way, where the performance can be radically different. The purpose of this new analysis methodology is to clarify the essential factors for good performance instead of a thorough comparison of different systems. Importantly, it opens a new way to evaluate disk-based systems analytically.

Improved Preprocessing. To reduce IO cost, all disk-based graph computation systems [3], [14] employ preprocessing to convert the input graph to an internal form on the disk. In preprocessing, GraphChi scans the entire input graph four times [3], which can be time-consuming for a big graph. In this paper, we propose a new preprocessing scheme (Section 4.1) that scans the graph only twice, and reduces the preprocessing time in our previous work [17] by 40%. Our experiments in Section 5.5 show that the preprocessing scheme of VENUS is superior to that of GraphChi [3], and although the preprocessing of X-Stream [14] does not need to sort edge lists, VENUS is still at least twice faster than X-Stream in terms of the total execution time.

Extensive experiments. We did extensive experiments on both large-scale real-world graphs and large-scale synthetic graphs to validate the performance of our approach. Our experiments look into several key performance factors to all disk-based single-machine systems including computation time, the effectiveness of main memory utilization, the amount of data read and write, and the number of shards generated. We found that VENUS is usually two to three times faster than GraphChi and X-Stream, two state-of-the-art disk-based systems, on a number of graph computing tasks including connected components, shortest paths, and alternating least squares. Its performance is also comparable to distributed platforms over clusters. For example, to run 5 iterations of the PageRank algorithm on a Twitter graph of 42 million vertices and 1.4 billion edges, Spark needs 8.1 minutes with 50 machines (100 CPUs) on Amazon's EC2 [18] and GraphChi spends 13 minutes using high-speed SSD [3], while VENUS only takes 5 minutes on one machine with an ordinary hard disk.

The rest of the paper is organized as follows. Section 2 gives an overview of VENUS, which includes a disk-based architecture, graph organization and storage, and an external computing model. Section 3 presents algorithms to substantialize our processing pipeline. Section 4 explains important considerations in implementing VENUS. We extensively evaluate VENUS in Section 5. Section 6 reviews more related work. We discuss the expressiveness and limitation of various computing models for graph computation in Section 7. Section 8 concludes the paper.

2 System Overview

VENUS is based on a new disk-based graph computation architecture, which supports a novel *vertex-centric streamlined processing* (VSP) computing model such that the graph is sequentially loaded and the update function is executed for each of vertices in parallel on the fly. To support the VSP model, we propose a graph storage scheme and an external graph computing model that coordinates the graph computation with CPU, memory, and disk access. By working together, the system significantly reduces the amount of data to be accessed, generates much fewer shards than the existing scheme [3], and effectively utilizes large main memory with provable performance guarantee.

2.1 Architecture Overview

The input is modeled as a directed graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Like existing work [19], [20], the user can assign a mutable vertex value to each vertex and define an arbitrary read-only edge value on each edge. Let (u, v) be a directed edge from vertex u to vertex v. Vertex u is called an in-neighbor of v, and v an out-neighbor of u. (u, v) is called an in-edge of v and an out-edge of u, and u and v are called the source and destination of edge (u, v) respectively.

Most graph tasks are iterative and vertex-centric in nature, and any update of a vertex value in each iteration usually involves only its in-neighbors' values. Once a vertex is updated, it will trigger the updates of its out-neighbors. This dynamic continues until convergence or certain conditions are met. The disk-based approach organizes the graph data into a number of *shards* on disk, so that each shard can fit in the main memory. Each shard contains all needed information for computing updates of a number of vertices. One iteration will execute all shards. Hence there is a huge amount of disk data to be accessed per iteration, which may result in extensive IOs and become a bottleneck of the

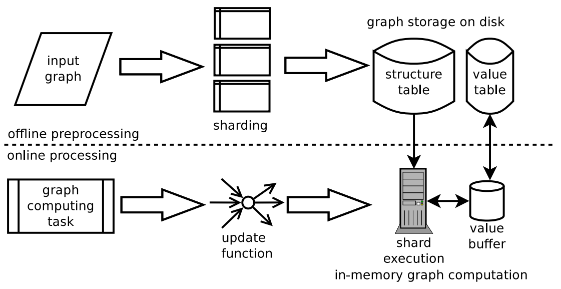


Fig. 1. The Architecture of VENUS

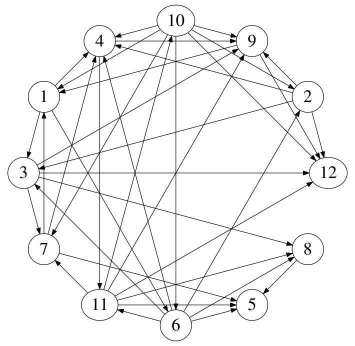


Fig. 2. Example Graph

disk-based approach. Therefore, a disk-based graph computation system needs to manage the storage, the use of memory, and CPU in an intelligent way to minimize disk access.

VENUS, its architecture depicted in Figure 1, makes use of a novel management scheme of disk storage and the main memory, in order to support vertex-centric streamlined processing. VENUS decomposes each task into two stages: offline preprocessing and online processing. In the offline preprocessing, VENUS reads the input graph file and constructs the graph storage on disk, which is organized as a number of shards. For each shard, the edges with their associated edge values are stored in the structure table, while the vertex data, including mutable vertex values, are kept in the value table. In the online processing, a graph computation task is defined using an update function. Then VENUS executes the update function for each vertex and manages the interaction between CPU, memory, and disk.

2.2 Vertex-Centric Streamlined Processing

VENUS enables vertex-centric streamlined processing (VSP) on our storage system, which is crucial in fast loading of graph data and rapid parallel execution of the update function. As we will show later, it has a superior performance with much less data transfer overhead. Furthermore, it is more effective in main memory utilization, as compared with other schemes. We will elaborate on this in Section 2.3. To support streamlined processing, we propose a new graph sharding method, a new graph storage scheme, and a novel external graph computing model. Let us now provide a brief overview of our sharding, storage, and external graph computing model.

Graph sharding. Suppose the graph is too big to fit in the main memory. Then how it is organized on disk will affect how it will be accessed and processed afterwards. VENUS splits the vertices set V into P disjoint intervals. Each interval defines a *g-shard* and a *v-shard*, as follows. The g-shard stores all the edges (and the associated attributes) with destinations in that interval. The v-shard contains all vertices in the g-shard which includes the source and destination of each edge. Edges in each g-shard are ordered by destination, where the in-edges (and their associated read-only attributes) of a vertex are stored consecutively as a *structure record*. There are $|V|$ structure records in total for the whole graph. The g-shard and the v-shard corresponding to the same vertex interval make a full shard. To illustrate the concepts of shard, g-shard, and v-shard, consider the graph with 12 vertices as shown in Figure 2. Suppose the vertices are divided into three intervals: $I_1 = [1, 4]$, $I_2 = [5, 8]$, and $I_3 = [9, 12]$. Then, the resulting shards, including g-shards and v-shards, are listed in Table 1.

TABLE 1
Sharding Example: VENUS

Interval	$I_1 = [1, 4]$	$I_2 = [5, 8]$	$I_3 = [9, 12]$
v-shard	$I_1 \cup \{6, 7, 9, 10\}$	$I_2 \cup \{1, 3, 10, 11\}$	$I_3 \cup \{2, 3, 4, 6\}$
g-shard	7,9,10 $\rightarrow$ 1	6,7,8,11 $\rightarrow$ 5	2,3,4,10,11 $\rightarrow$ 9
	6,10 $\rightarrow$ 2	1,10 $\rightarrow$ 6	11 $\rightarrow$ 10
	1,2,6 $\rightarrow$ 3	3,10,11 $\rightarrow$ 7	4,6 $\rightarrow$ 11
	1,2,6,7,10 $\rightarrow$ 4	3,6,11 $\rightarrow$ 8	2,3,9,10,11 $\rightarrow$ 12
$S(I)$	6	1	2
	7	3	3
	9	10	4
	10	11	6

In practice, all g-shards are further concatenated to form the structure table, i.e., a stream of structure records (Figure 3). Such a design allows executing vertex update on the fly, and is crucial for VSP. Using this structure, we do not need to load the whole subgraph of vertices in each interval before execution as in GraphChi [3]. Observing that more shards usually incur more IOs, VENUS aims to generate shards as few as possible. To this end, a large interval is preferred provided that the associated v-shard can be loaded completely into the main memory, and there is no size constraint on the g-shard. Once the vertex values of vertices in a v-shard is loaded and then held in the main memory, VENUS can readily execute the update function for all vertices in the interval with only "one sequential scan" over the corresponding g-shard. We will discuss how to load and update vertex values for vertices in each v-shard in Section 3.

Graph storage. We propose a new graph storage that aims to reduce data access. Recall that the graph data consists of two parts, the read-only structure records, called *structure data*, and the mutable vertex values, called *value data*. We observe that in one complete iteration, the entire structure data needs to be scanned only once, while the value data usually needs to be accessed multiple times, because a vertex value is involved in each update of its out-neighbors. This suggests us to organize the structure data as consecutive pages, and it should be separated from the value data. As such, the access of the massive volume structure data can be done highly efficiently with one sequential scan (sequential IOs). Specifically, we employ an operating system file, called the *structure table*, which is optimized for sequential scan, to store the structure data.

Note that the updates and repeated reads over the value data can result in extensive random IOs. To cope with this, VENUS deliberately avoids storing a significant amount of structure data

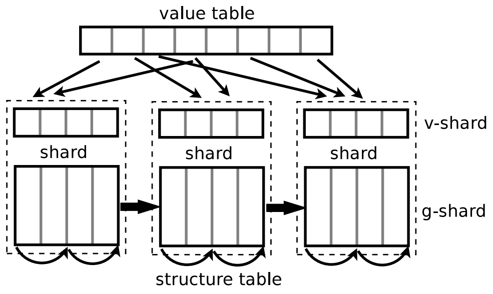


Fig. 3. Vertex-Centric Streamlined Processing

into the main memory, as required in GraphChi [3], and instead caches value data in the main memory as much as possible. VENUS stores the value data in a disk table, which we call the *value table*. The value table is implemented as a number of consecutive disk pages, containing $|V|$ fixed length *value records*, each per vertex. For simplicity of presentation, we assume all value records are arranged in ascending order (in terms of vertex ID).

External computing model. Given the above description of graph sharding and storage, we are ready to present our graph computing model which processes the incoming stream of structure records on the fly. Each incoming structure record is passed for execution as the structure table is loaded sequentially. A higher execution manager is deployed to start new threads to execute new structure records in parallel, when possible. A structure record is removed from main memory after its execution, so as to make room for next processing. On the other hand, the required vertex values of the active shard are obtained based on v-shard, and are buffered in main memory throughout the execution of that shard. As a result, the repeated access of the same vertex value can be done in the buffer even for multiple shards. We illustrate the above computing process in Figure 3.

We use the graph in Figure 2 to illustrate and compare the processing pipelines of VENUS and GraphChi. The sharding structures of VENUS are shown in Table 1, and those for GraphChi are in Table 2 where the number of shards is assumed to be 4 to reflect the fact that GraphChi usually uses more shards than VENUS. To begin, VENUS first loads v-shard of I_1 into the main memory. Then we load the g-shard in a streaming fashion from disk. As soon as we are done loading all the in-edges of vertex 1 (which include edges $(7, 1)$, $(9, 1)$, and $(10, 1)$), we can perform the value update on vertex 1, and at the same time, we load the in-edges of vertices 2, 3, and 4 in parallel. In contrast, to perform computation on the first interval, GraphChi needs to load all related edges (shaded edges in the table), which include all the in-edges and out-edges for the interval. This means that for processing the same interval, GraphChi requires more memory than VENUS. So under the same memory constraint, GraphChi needs more (and smaller) shards. More critically, because all in-edges and out-edges must be loaded before computation can start, GraphChi cannot parallelize IO operations and computations like VENUS.

2.3 Analysis

We now compare our proposed VSP model with two popular single-PC graph computing models: the *parallel sliding windows* model (PSW) of GraphChi [3] and the *edge-centric processing* model (ECP) of X-Stream [14]. Specifically, we look at three evaluation criteria: 1) the amount of data transferred between disk and the main memory per iteration; 2) the number of shards generated; and 3) the adaptation to large memory.

There are strong reasons to develop our analysis based on the first criterion, i.e., the amount of data transfer: (i) it is fundamental and applicable for various types of storage systems, including magnetic disk and solid-state disk (SSD), and various types of memory hierarchies such as on-board cache/RAM and RAM/disk; (ii) it can be used to derive IO complexity of disk-based algorithms as in Section 3.3; and (iii) it helps examine other criteria, such as the number of shards and large memory adaptation. We summarize the results in Table 4, and show the details of our analysis below. Note that the second criterion is related closely to IO complexity, and the third criterion examines the utilization of memory.

TABLE 2
Sharding Example: GraphChi

Interval	$I_1 = [1, 3]$	$I_2 = [4, 6]$	$I_3 = [7, 9]$	$I_4 = [10, 12]$
Shard	$1 \to 3$	$1 \to 4{,}6$	$2 \to 9$	$2 \to 12$
	$2 \to 3$	$2 \to 4$	$3 \to 7{,}8{,}9$	$3 \to 12$
	$6 \to 2{,}3$	$6 \to 4{,}5$	$4 \to 9$	$4 \to 11$
	$7 \to 1$	$7 \to 4{,}5$	$6 \to 8$	$6 \to 11$
	$9 \to 1$	$8 \to 5$	$10 \to 7{,}9$	$9 \to 12$
	$10 \to 1{,}2$	$10 \to 4{,}6$	$11 \to 7{,}8{,}9$	$10 \to 12$
		$11 \to 5$		$11 \to 10{,}12$

TABLE 3
Notations

Notation	Definition
n, m	$n = \|V\|, m = \|E\|$
$gs(I)$	g-shard of interval I
$vs(I)$	v-shard of interval I
$S(I)$	$\{u \notin I \| (u, v) \in gs(I)\}$
δ	$\delta = \sum_I \|S(I)\|/m$
P	number of shards
M	size of RAM
C	size of a vertex value record
D	size of one edge field in a structure record
B	size of a disk block accessed by a unit IO

For easy reference, we list the notation in Table 3. For our VSP model, V is split into P disjoint intervals. Each interval I defines a g-shard and a v-shard. A g-shard is defined as

$$gs(I) = \{(u, v)|v \in I\},$$

and a v-shard is defined as

$$vs(I) = \{u|(u, v) \in gs(I) \vee (v, u) \in gs(I)\}.$$

Note that $vs(I)$ can be split into two disjoint sets I and $S(I)$, where $S(I) = \{u \notin I|(u, v) \in gs(I)\}$. Note that

$$\sum_I |S(I)| \leq \sum_I |gs(I)| = m.$$

Let δ be a scalar between 0 and 1 such that

$$\sum_I |S(I)| = \delta m.$$

It can be seen that

$$\sum_I |vs(I)| = \sum_I |S(I)| + \sum_I |I| = \delta m + n.$$

TABLE 4
Analysis of graph computing models

category	PSW	ECP	VSP
Data read	$Cn+2(C+D)m$	$Cn+(C+D)m$	$C(n+\delta m)+Dm$
Data write	$Cn+2(C+D)m$	$Cn+Cm$	Cn
No. of shard	$\frac{Cn+2(C+D)m}{M}$	$\frac{Cn}{M}$	$\frac{C(n+\delta m)}{M}$

Let C be the size of a vertex value record, and let D be the size of one edge field in a structure record. We use B to denote the size of a disk block accessed in a unit IO. According to [14], both SSDs and hard disks are saturated with sequential access of size 16 MB, so we suppose B equals to 16 MB.

Data transfer. For each iteration, VENUS loads all g-shards and v-shards from disk, which needs Dm and $C(n+\delta m)$ data read in total. After the computation is done, VENUS writes v-shards back to disk which incurs Cn data write. Note that g-shards are read-only.

Unlike VENUS where each vertex can access the values of its neighbors through v-shard, GraphChi accesses such values from the edges. So the data size of each edge in GraphChi is $(C+D)$. For each iteration, GraphChi processes one shard at a time. The processing of each shard is split into three steps: (1) loading a subgraph from disk; (2) updating the vertices and edges; (3) writing the updated values to disk. In steps 1 and 3, each vertex will be loaded and written only once which incurs Cn data read and write. For edges data, in the worst case, each edge is accessed twice (once in each direction) in step 1 which incurs $2(C+D)m$ data read. If the computation updates edges in both directions in step 2, the size of data write of edges in step 3 is also $2(C+D)m$. So the data read and write in total are both $Cn+2(C+D)m$.

In the disk-based engine of X-Stream, one iteration is divided into (1) merged scatter/shuffle phase and (2) gathering phase. In phase 1, X-Stream loads all vertex value data and edge data, and for each edge it writes an update to disk. Since updates are used to propagate values passed from neighbors, we suppose the size of an update is C. So for phase 1, the size of read is $Cn+Dm$ and the size of write is Cm. In phase 2, X-Stream loads all updates and updates each vertex, so the size of read is Cm and the size of write is Cn. So for one full pass over the graph, the size of read is $Cn+(C+D)m$ and the size of write is $Cn+Cm$ in total.

Number of shards. For interval I, VENUS only loads the v-shard $vs(I)$ into memory and the g-shard $gs(I)$ is loaded in a streaming fashion. So the number of shards is determined by the total size of v-shards and we have $P=\frac{C(n+\delta m)}{M}$. In contrast, GraphChi loads both vertex value data and edge data for each interval, so the number of shards P in GraphChi is $\frac{Cn+2(C+D)m}{M}$. In X-Stream, edges data are also loaded in a streaming fashion, so the number of intervals is $P=\frac{Cn}{M}$.

We can see that the number of shards constructed in VENUS is always smaller than that in GraphChi. In Section 3, we will show that the smaller of the number of shards, the lower of IO complexity.

Adaptation to large memory. As analyzed above, for our VSP model, the size of data read in one iteration is $C(n+\delta m)+Dm$. So one way to improve performance is to decrease δ. Here we show that δ does decrease as the size of available memory increases, which implies that VENUS can exploit the main memory effectively. Suppose the memory size is M, and the vertex set V is split into P intervals $I_1, I_2, \ldots, I_P$, where $vs(I_i) \le M$ for $i=1,\ldots,P$. Then, by definition, $\delta m=\sum_{i=1}^{P}|S(I_i)|$. Now, consider a larger memory size M' such that $M' \ge |vs(I_1)|+|vs(I_2)| \ge M$. Under the memory size M', we can merge interval I_1 and I_2 into I_t, because $|vs(I_t)| \le |vs(I_1)|+|vs(I_2)| \le M'$. Suppose $\delta' m=|S(I_t)|+\sum_{i=3}^{P}|S(I_i)|$. By the definition of $S(I)$, it can be shown that $S(I_t) \subseteq S(I_1) \cup S(I_2)$, and thus $|S(I_t)| \le |S(I_1)|+|S(I_2)|$. Therefore we have $\delta' \le \delta$, which means as M increases, δ becomes smaller. When $M \ge Cn$, we have $P=1$ where $\delta=0$. In such a single shard case, the data size of read reaches the lower bound $Cn+Dm$.

3 Storage and Computation

In this section, we discuss the full embodiment of our vertex-centric streamlined processing model, including the details of our graph storage design, the online computing state management, and the main memory usage. It consists of two IO-friendly algorithms with different flavors and IO complexities in implementing the processing of Section 2. Note that the IO results here are consistent with the data transfer size results in Section 2 because the results here are obtained with optimization specialized for disk-based processing to transfer the same amount of data. Since the computation is always centered on an active shard, the online computing state mainly consists of the v-shard values that belong to the active shard.

Our first algorithm materializes all v-shard values in each shard, which supports fast retrieval during the online processing. However, in-time view update on all such views is necessary once the execution of each shard is finished. We employ an efficient scheme to exploit the data locality in all materialized views. And this scheme shares a similar spirit with the parallel sliding window of [3], with quadratic IO performance in P, the number of shards. In order to avoid the overhead of view maintenance at run time, our second algorithm applies "merge-join" to construct all v-shard values on-the-fly, and updates the active shard only. The second algorithm has an IO complexity linear in P. Finally, as the RAM becomes large, the two algorithms adapt to the memory scaling with less sharding overhead, and finally the two algorithms automatically work in the in-memory mode to seamlessly integrate the case when the main memory can hold all vertex values.

3.1 Physical Design and The Basic Procedure

The tables. The value table is implemented as a number of consecutive disk pages, containing $|V|$ fixed-length value records, each per vertex. For the ease of presentation, we assume all value records are arranged in the ascending order of their IDs in the table. For an arbitrary vertex v, the disk page containing its value record can be loaded in $O(1)$ time. Specifically, the number of the value records in one page, N_B, is $N_B=\lfloor \frac{B}{C} \rfloor$, where B is the page size and C is the size of the vertex value. Thus, the value record of v can be found at the slot $(v \bmod N_B)$ in the $\lfloor \frac{v}{N_B} \rfloor$-th page.

Note that the edge attributes will not change during the computation. We pack the in-edges of each vertex v and their associated read-only attributes into a variable length *structure record*, denoted as $R(v)$, in the structure table. Each structure record $R(v)$ starts with the number of in-edges of vertex v, followed by the list of source vertices of in-edges and the read-only attributes. One structure record usually resides in one disk page and can span

Procedure ExecuteVertex(v, R(v), VB, I)

```
input : vertex $v$, structure record $R(v)$, value buffer VB, and
        interval $I$.
output: the updated record of $v$ in the value table.
1 foreach $s \in R(v)$ do
2     let $Q$ be the $\lfloor \frac{s}{N_B} \rfloor$-th page of the value table;
3     if $s \in I \wedge Q \notin$ VB then
4         Pin $Q$ into VB;
5 let val be the value record of $v$ in the value table;
6 val ← UpdateVertex($R(v)$, VB);
```

Algorithm 1: Execute One Iteration with Views

```
1  let $I$ be the first interval;
2  load $view(I)$ into the map of VB;
3  foreach $R(v)$ in the structure table do
4      if $v \notin I$ then
5          foreach internal $J \neq I$ do
6              $view(J)$.UpdateActiveWindowToDisk();
7          unpin all pages and empty the map, in VB;
8          set $I$ to be the next interval;
9          load $view(I)$ into the map of VB;
10     ExecuteVertex($v, R(v)$, VB, $I$)
```

multiple disk pages for vertices of large degrees. Hence, there are $|V|$ such records in total. As an example, for the graph in Figure 2, the structure record $R(3)$ of vertex 3 contains incoming vertices 1, 2, and 6 and their attributes.

The basic procedure. In VENUS, there is a basic execution procedure, namely, Procedure ExecuteVertex, which represents the unit task that is being assigned and executed by multiple cores in the computer. Moreover, Procedure ExecuteVertex also serves as a common routine that all our algorithms are built upon it, where the simplest one is the in-memory mode to be explained below.

Procedure ExecuteVertex takes a vertex $v \in I$, the structure record $R(v)$, the *value buffer* VB (call-by-reference), and the current interval I as its input. The value buffer VB maintains all latest vertex values of v-shard $vs(I)$ of interval I. In VB, we use two data structures to store vertex values, i.e., a *frame table* and a *map*. Note that $vs(I)$ can be split into two disjoint vertex sets I and $S(I)$. The frame table maintains all pinned value table pages of the vertices within interval I; the map is a dictionary of vertex values for all vertices within $S(I)$. Therefore, VB supports the fast look-up of any vertex value of the current v-shard $vs(I)$. Procedure ExecuteVertex assumes the map of VB already includes all vertex values for $S(I)$. How to realize this is addressed in Section 3.2. Suppose vertex s is an in-neighbor of v, if the value table page of s has not been loaded into the frame table yet, we pin the value table page of s at Line 4. After all required vertex values for $R(v)$ are loaded into memory, we execute the user-defined function, `UpdateVertex()`, to update the value record of v at Line 6. This may implicitly pin the value table page of v. All pages will be kept in the frame table of VB for later use, until an explicit call to unpin them.

Consider the graph in Figure 2 and its sharding structures in Table 1. Suppose $I = I_1$. For the value buffer VB, the frame table contains value table pages of vertices 1, 2, 3, and 4 in I_1, and the map contains vertex values of vertices 6, 7, 9, and 10 in $S(I_1)$.

We can now explain our in-memory mode. It requires that the entire value table be held in the main memory and hence only one shard exists. In this mode, the system performs sequential scan over the structure table from disk, and for each structure record $R(v)$ we encountered, an executing thread starts Procedure ExecuteVertex for it on the fly. Note that in Procedure ExecuteVertex, I includes all vertices in V and the map in VB is empty. Upon the end of each call of Procedure ExecuteVertex, $R(v)$ will be no longer needed and be removed immediately from the main memory for space-saving. So we stream the processing of all structure records in an iteration. After an explicitly specified number of iterations have been done or the computation has converged, we can unpin all pages in VB and terminate the processing. To overlap disk operations as much as possible, all disk accesses over structure table and value table are done by concurrent threads, and multiple executing threads are concurrently running to execute all subgraphs.

3.2 The Algorithms for Accessing Shards

When all vertex values cannot be held in main memory, the capacity of VB is inadequate to buffer all value table pages. The in-memory mode described above cannot be directly applied in this case, otherwise there will be seriously system thrashing. Based on the discussion of Section 2.2, we split V into P disjoint intervals, such that the vertex values of each v-shard can be entirely buffered into main memory.

In this case, we organize the processing of a single shard to be extendible in terms of multiple shards. The central issue here is how to manage the computing states of all shards to ensure the correctness of processing. This can be further divided into two tasks that must be fulfilled in executing each shard:

- constructing the map of VB so that the active shard can be executed based on Procedure ExecuteVertex according to the previous discussion;
- synchronizing intermediate results to disk so that the latest updates are visible to any other shard to comply with the asynchronous parallel processing [3].

Note that these two tasks are performed based on the v-shard and the value table. In summary, the system still performs sequential scan over the structure table from disk, and continuously loads each structure record $R(v)$ and executes it with Procedure ExecuteVertex on the fly. Furthermore, the system also monitors the start and the end of the active shard, which triggers a call to finish the first and/or the second tasks. This is the framework of our next two algorithms.

The algorithm using dynamical view. Our first algorithm materializes all v-shard values as a *view* for each shard, which is shown in Algorithm 1. Specifically, we associate each interval I with $view(I)$ which materializes all vertex values of vertices in $S(I)$. Thus the first task is to load this view into the map of VB, which is done for Line 2 or Line 9. Then, at the time when we finish the execution of an active shard and before we proceed to the next shard, we need to update the views of all other shards to reflect any changes of vertex values that can be seen by any other shard (Line 5 to Line 6). To do this efficiently, we exploit the data locality in all materialized views.

Specifically, the value records of each view are ordered by their vertex ID. So in every view, say the i-th view for the i-th shard, all the value records for the j-th interval, $i \neq j$, are stored

Algorithm 2: Execute One Iteration with Merge-Join

```
1 let $I$ be the first interval;
2 join $S(I)$ and the value table to polulate the map of VB;
3 foreach $R(v)$ in the structure table do
4     if $v \notin I$ then
5         unpin all pages and empty the map, in VB;
6         set $I$ to be the next interval;
7         join $S(I)$ and the value table to populate the map of VB;
8     ExecuteVertex($v, R(v), \mathrm{VB}, I$)
```

consecutively. And more importantly, the value records in the $(j+1)$-th interval are stored immediately after the value records for the j-th interval. Therefore, similar to the parallel sliding window of [3], when the active shard is shifted from an interval to the next, we can also maintain an *active sliding window* over each of the views. And only the active sliding window of each view is updated immediately after we finish the execution of an active shard (Line 6).

Consider the example in Figure 2 and Table 1. For computation on interval I_2, loading the vertex values in $S(I_2)$ can be easily done with one sequential disk scan over $view(I_2)$, because the latest vertex values are already stored in $view(I_2)$. After computation, we need to propagate the updated value records to other intervals. In this example, we update those vertex values in the active sliding windows of $view(I_1)$ and $view(I_3)$ (shaded cells in Table 1).

The algorithm using merge-join. Our second algorithm uses merge-join over the v-shard and the value table. Its main advantage is without the overhead to maintain all views at run time. It is shown in Algorithm 2. Specifically, we join $S(I)$ for each interval I with the value table to obtain all vertex values of $S(I)$. Since both $S(I)$ and the value table are sorted by the vertex ID, it is easy to use a merge-join to finish that quickly. The join results are inserted into the map of VB at Line 2 and Line 7. All vertex values are directly updated in the value table, and any changes of vertex values are immediately visible to any other shard.

Again, we consider the example in Figure 2 and Table 1. Suppose that we want to update interval I_1. First, we need to load $S(I_1) = \{6, 7, 9, 10\}$ into the map of VB. To load $S(I_1)$, we use a merge-join over the vertex table and $S(I_1)$. Since the vertex table and $S(I_1)$ are both sorted by vertex ID, we just need one sequential scan over the vertex table. The updated values of vertices in I_1 are written to the value table directly which incurs only sequential IOs.

Finally, as the RAM becomes large enough to hold the complete value table, only one shard and one interval for all vertices presents. The view/merge-join is no longer needed. Both algorithms automatically work in the in-memory mode.

3.3 IO Analysis

To compare the capabilities and limitations of the two algorithms, we look at the IO costs of performing one iteration of graph computation using the theoretical IO model [21]. In this model, the IO cost of an algorithm is the number of block transfers from disk to main memory plus the number of non-sequential seeks. So the complexity is parametrized by the size of block transfer, B.

TABLE 5
Big-O bounds in the IO model

System	# Read IO	# Write IO
GraphChi [3]	$\frac{Cn+2(C+D)m}{B} + P^2$	$\frac{Cn+2(C+D)m}{B} + P^2$
X-Stream [14]	$\frac{Cn+(C+D)m}{B}$	$\frac{Cn}{B} + \frac{Cm}{B} \log_{\frac{M}{B}} P$
Alg. 1	$\frac{C(n+\delta m)+Dm}{B}$	$\frac{C(n+\delta m)}{B} + P^2$
Alg. 2	$P\frac{Cn}{B} + \frac{Dm}{B}$	$\frac{Cn}{B}$

For Algorithm 1, the size of data read is $C(n + \delta m) + Dm$ (Table 4). Since loading does not require any non-sequential seeks, the number of read IOs is $\frac{C(n+\delta m)+Dm}{B}$. On the other hand, to update all v-shards data, the number of block transfers is $\frac{C(n+\delta m)}{B}$. In addition, in the worst case, each interval requires P non-sequential seeks to update the views of other shards. Thus, the total number of non-sequential seeks for a full iteration has a cost of P^2. So the total number of write IOs of Algorithm 1 is $\frac{C(n+\delta m)}{B} + P^2$.

For Algorithm 2, the number of read IOs can be analyzed by considering the cost of merge-join for P intervals, and then adding to this the cost of loading the structure table. The cost of merge-join for each interval is $\frac{Cn}{B}$. The size of structure table is Dm. Thus, the total number of read IOs is $P\frac{Cn}{B} + \frac{Dm}{B}$. For interval I, the cost of updating the value table is $\frac{C|I|}{B}$. Hence, the total number of write IOs is $\sum_I \frac{C|I|}{B} = \frac{Cn}{B}$.

Table 5 shows the comparison of GraphChi, X-Stream, and our algorithms. We can see that the IO cost of Algorithm 1 is always less than GraphChi. Also, when P is small, the numbers of read IOs of Algorithm 1 and Algorithm 2 are similar, but the number of write IOs of Algorithm 2 is much smaller than that of Algorithm 1. These results can guide us in choosing proper algorithms for different graphs.

4 System Implementation

In this section, we explain several important considerations in the offline preprocessing and online processing of our system, VENUS.

4.1 Offline Preprocessing

Structure Table. The offline processing converts the input graph into a number of shards, which are stored in the structure table and the value table. The design of the value table is straightforward. We will explain the structure table as follows. As described in Section 2, all g-shards are concatenated to form the structure table which contains $|V|$ structure records. The structure table is in binary format which allows fast construction and access. To support different graph computation tasks, where the associated edge values can vary, we use an *adjacency file* to store edges and an *attribute file* for the associated edge values.

The adjacency file stores the neighbors for each vertex with a list. All vertices with corresponding lists are arranged in the ascending order of the vertex ID. For each vertex, the list starts with a number for the degree. Then, the rest of the list consists all vertex IDs of the neighbors. Each vertex only has one list for the in-neighbors by default, and it is optional to store another list for the out-neighbors; in the attribute file, all edge values are just stored as a flat array of the user-defined type.

Two-Step Preprocessing. To construct the structure table from a given graph in various formats such as edge list, adjacency list, or matrix format, we design the two-step preprocessing based on the external merge sort algorithm which is efficient in IO and with modest memory requirement (see Table 6).

In the first step, we maintain t buffers, each in M/t MB size, where M is the user-given memory budget in MB. We read the input graph data sequentially and add each encountered edge into some buffer which is not fully occupied. If a buffer is full, we sort the edges in the buffer based on the edge destination and write these edges into an intermediate file, which we denote as *chunk*, to empty that buffer. We repeat this process until the whole input file is turned into a number of such sorted chunks. Note that we can greatly accelerate this process with multiple threads: we assign one thread for each fully occupied buffer to do the sorting and construction of the chunk, while there is one main thread to read the input data and distribute the input edges. Therefore, at most t threads are needed. In this paper, we set t as 3.

The second step performs a k-way merge on all chunks resulted from the first step to construct the structure table directly. In summary, the preprocessing scans the input graph with two passes in total. Therefore, its cost is proportional to the graph size. We will further illustrate this cost in our experiments in Section 5.5.

4.2 Online Processing

Sharding. Like GraphChi [3] and X-Stream [14], VENUS allows user to explicitly specify a budget to limit the usage of main memory. It is interesting to notice that VENUS can perform sharding with various memory budgets based on the same structure table and value table. Its benefit is that we can test VENUS under different memory budgets with just one and the same preprocessing. This is done before executing the update function during the online processing. In contrast, GraphChi has to perform preprocessing each time when the memory budget changes.

In detail, VENUS scans the structure table and splits the vertices set V into P disjoint intervals according to the memory budget for the value buffer VB. Then, VENUS initializes the mutable vertex values in VB and also $S(I)$ and $view(I)$ for each interval I. After that, VENUS starts to execute the update function for each vertex, where the details are already explained in Section 3. Specifically, we spend half of the main memory budget for the frame table in VB, which is managed based on the LRU replacement strategy; and another $\frac{1}{4}$ of the main memory budget is for the map in VB, leaving the remaining memory to store auxiliary data. Next, we present other optimizations that speed up our computation.

Scheduler. In graph computation, the computation on some vertices may converge faster than others. For example, in the single-source shortest path problem (SSSP), we only need to update the shortest path to a vertex when the shortest paths to some of its neighbors change. It is useful if we can schedule the vertices, so we can avoid the execution of the update function for unscheduled vertices and also avoid loading structure records of these vertices from disk. This feature is called *selective scheduling* [3], [20]. In VENUS, we implement a *scheduler* to support this feature. The scheduler allows a vertex to choose some of its neighbors and add them to the scheduler in the update function. In the case of SSSP, only when a vertex changes the shortest path from the source to itself in the update function, the vertex will add all of its neighbors to the scheduler. In order to reduce the IO cost as much as possible, VENUS is further optimized to avoid accessing their structure records from disk for those vertices not in the scheduler. In detail, VENUS represents the scheduler as a bitset to ensure that each vertex occupies only one bit in memory. In the preprocessing, we split the structure table into small blocks of size 64 MB and VENUS will skip the blocks which do not contain scheduled vertices.

Multi-Threading. We use multi-threading to let VENUS overlap the IO operations and the execution of the update function to speed up the overall computation time. In detail, VENUS loads the structure table from disk using a dedicated IO thread. In addition, there are multiple computing threads to parallelize the execution of the update function for large amount of vertices. Each time the IO thread loads a block of the structure table into memory to form in-memory vertex data, which is a vector of multiple vertices and their structure records. The multiple computing threads will simultaneously consume these in-memory vertex data by running the update function among these vertices in parallel using the OpenMP library [22]. In this way, we can leverage the power of multi-core architecture of a single machine.

5 Performance Evaluation

In this section, we evaluate our system VENUS and compare it with two most related state-of-the-art systems, GraphChi [3] and X-Stream [14]. GraphChi uses the parallel sliding window model and is denoted as PSW in all figures. X-Stream employs the edge-centric processing model and thus is denoted as ECP. Our system is built on the vertex-centric streamlined processing (VSP) model which is implemented with two algorithms: Algorithm 1 materializes vertex values in each shard for fast retrieval during the execution, which is denoted as VSP-I; Algorithm 2 applies merge-join to construct all vertex values on the fly, which is denoted as VSP-II. The two algorithms use the same vertex-centric update function for a graph task and an input parameter indicates which algorithm should be used. All three systems are coded in C++. We ran each experiment three times and reported the averaged execution time. All algorithms are evaluated using hard disk, so we do not include TurboGraph [15] due to its requirement of SSD drive on the computer.

All experiments are conducted on a commodity machine with Intel i7 quad-core 3.4 GHz CPU, 16 GB RAM, and 4 TB hard disk, running Linux. Note that GraphChi and VENUS use Linux system calls to access data from disk, where the operating system caches data in its *pagecache*. This allows GraphChi and VENUS to take advantage of extra main memory in addition to the memory budget. On the other hand, X-Stream uses direct IO and does not benefit from this. Therefore, for the sake of fairness, we use pagecache-management[1] to disable pagecache in all our experiments.

We mainly examine three important aspects of a system which are key to its performance: 1) computation time; 2) the amount of data read and write; and 3) the number of shards. We explain the effectiveness of main memory utilization of VENUS based on these results. We experiment over 4 large real-world graph datasets, twitter-2010 [23], clueweb12 [24][2], Netflix [28], and

1. https://code.google.com/p/pagecache-mangagement/
2. This dataset is obtained from the Laboratory for Web Algorithmics [25] and decompressed using the WebGraph framework [26], [27]

TABLE 6
Graph datasets and preprocessing time

			Preprocessing Time (sec.)		
Dataset	$\|V\|$	$\|E\|$	GraphChi	X-Stream	VENUS
twitter-2010	41.7 M	1.4 B	817.7	258.8	617.6
clueweb12	978.4 M	42.5 B	24705.4	3708.3	15545.3
Netflix	0.5 M	99.0 M	287.5	27.1	66.7
KDD-Cup	1.6 M	252.8 M	571.2	66.6	192.2
Synthetic-4m	4 M	54.37 M	25.4	12.0	17.8
Synthetic-6m	6 M	86.04 M	39.2	17.9	27.1
Synthetic-8m	8 M	118.58 M	55.5	20.3	35.5
Synthetic-10m	10 M	151.99 M	69.4	25.9	47.6

Yahoo! Music user ratings used in KDD-Cup 2011 [29] as well as synthetic graphs. We use the SNAP graph generator[3] to generate 4 random power-law graphs, with increasing number of vertices, where the power-law degree exponent is set as 1.8. The data statistics are summarized in Table 6. We consider five graph computing tasks in the following categories:

1) **Graph mining**: Computation of weakly connected components (WCC) [16], community detection (CD) [16], and single-source shortest paths (SSSP);
2) **Sparse matrix-vector multiplication (SpMV)** : Computation of PageRank [30];
3) **Collaborative filtering**: Computation of alternating least squares (ALS) [31].

5.1 Exp-1: PageRank on the twitter-2010 graph

The first experiment runs 10 iterations of PageRank on the twitter-2010 graph. We compare the four algorithms (PSW, ECP, VSP-I, VSP-II) under various memory budgets from 0.5 GB to 8 GB.

Overall time. The results of processing time are reported in Figure 4(a), where we can see that VSP is up to 6x faster than PSW and ECP. For example, in the case that the memory budget is 8 GB, PSW takes 3,257.3 seconds and ECP takes 3,862.1 seconds. However, VSP-I and VSP-II just take 574.5 seconds and 574.1 seconds respectively. To further illustrate the efficiency of VSP, we also examine various performance factors including preprocessing, sharding, data access, and random IOs, as shown below.

Effectiveness of streamlined processing. To see the benefit of the streamlined processing of VENUS, we compare PSW and VSP in terms of the overall waiting time before executing a next shard. Figure 4(b) shows the waiting time in this experiment. For PSW, it includes the loading and sorting time of each memory shard; for VSP, it includes the time to execute unpin calls, view updates, and merge-join operations. Note that the time of scanning the structure table is evenly distributed among processing all vertices, and is not included here. It can be observed that PSW spends a significant amount of time for processing the shards before execution. In contrast, such waiting time for VSP is much smaller. This is due to that VSP allows to execute the update function while streaming in the structure data. For example, in the case that the memory budget is 8 GB, PSW takes 1,576.6 seconds. However, VSP-I and VSP-II just take 347.5 and 346.5 seconds respectively. Note that about the half share of the processing time of PSW is spent here, which spends far more time than our algorithms.

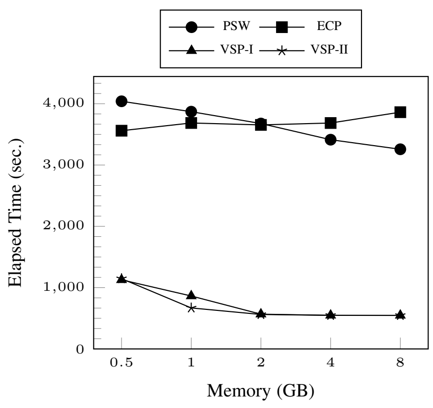
3. http://github.com/snap-stanford/snap

Adaptation to large memory. As described in Section 2.3, VENUS can exploit the main memory effectively: when the size of available memory increases, δ decreases. This can further reduce the number of shards and the data size of read in VENUS. In this experiment, as the memory budget increases from 0.5 GB to 8 GB, the value of δ becomes smaller and smaller as 0.164, 0.097, 0.025, 0, and 0. As a consequence, VSP also generates significantly smaller number of shards than PSW, as shown in Figure 4(c). For example, in the case that the memory budget is 0.5 GB and 1 GB, PSW generates 90 and 46 number of shards, respectively. And these numbers for our algorithms are 22 and 9. This is because VSP spends the memory budget on the value data of a v-shard, while the space needed to keep related structure data in memory is minimized. When the memory budget is 4 GB, VSP can hold all vertex values in main memory and run in its in-memory mode. Note that as we increase the memory budget to 8 GB, the processing time of VSP remains the same as shown in Figure 4(a), because the time used to read the structure table does not change. However, if we are given a machine with much larger RAM that VENUS can also cache the structure table in memory, then VENUS will achieve much better performance. We do not elaborate on such case as it is out of the focus of this paper.

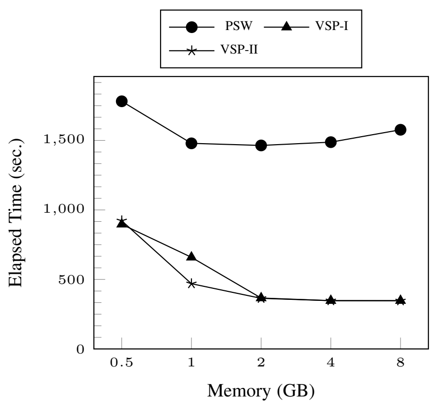
Reducing data transfer. To see the improvement of accessing disk data, Figure 4(e) and Figure 4(f) show the amount of data write and read, respectively. We observe that the data size written/read to/from disk is much smaller in VSP than in the other systems. Specifically, under the memory budget of 8 GB, PSW has to write 103.9 GB data to disk, and read 334.3 GB data from disk in total. These numbers for ECP are 121.0 GB and 306.3 GB, which are also very large and become a significant setback of ECP in its edge-centric streamlined processing model. In sharp contrast, VSP only writes 2.5 GB, which is 40X and 50X smaller than PSW and ECP, respectively. In terms of data size of read, VSP reads 74.6 GB data. The superiority of VSP in data access is mainly due to the separation of structure data and value data and the effective utilization of main memory.

Comparison of VSP-I and VSP-II. Our two algorithms have different processing time when the memory budget is small. Figure 4(d) provides one reason for this in terms of the IO costs of VSP-I and VSP-II when the memory budget is below 1 GB. The number of write IOs for VSP-II is always smaller than that of VSP-I which agrees with the analysis in Section 3.3. When the memory budget is 1 GB, VSP-II is faster and incurs fewer IOs than VSP-I because it does not need to maintain the materialized view in executing shards. When we further limit the memory budget as 0.5 GB, the number of read IOs for VSP-I increases slowly since it is bounded by $O(\frac{C(n+\delta m)+Dm}{B})$. However, the number of read IOs for VSP-II increases much faster since it is bounded by $P\frac{Cn}{B}+\frac{Dm}{B}$ and linear in P. In general, we suggest to use VSP-II except in the case when the memory budget is much smaller than the size of vertex data and P is larger than $\frac{m}{n}$ (e.g., we should choose VSP-I when the memory budget is 0.5 GB).

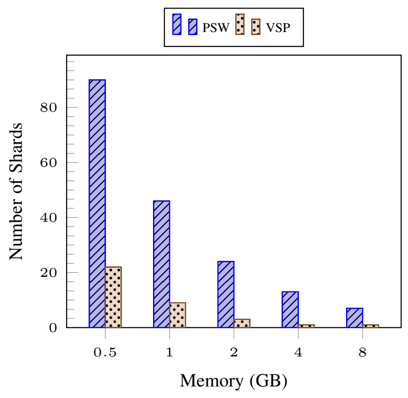
Better utilization of multi-core. To leverage the multi-core architecture of modern computers, both PSW and VSP can use multi-threading to accelerate the execution of the update function of each vertex. With the same memory budget (4 GB), Figure 5(a) shows the performance of PSW and VSP improves by 4% and 31% respectively, as we increase the number of threads from 1 to 8. Unfortunately, the performance does not improve linearly as



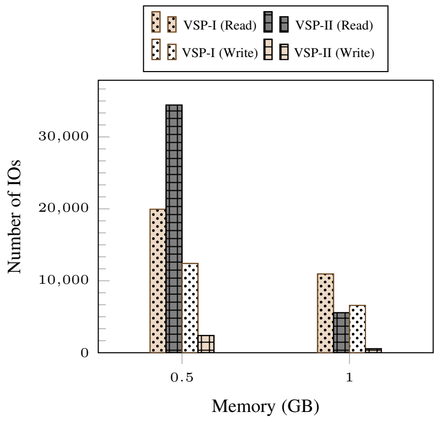
(a) Overall Time



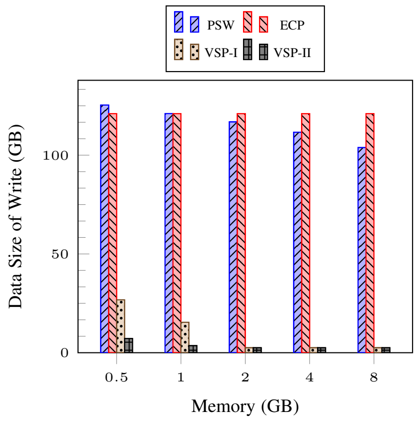
(b) Waiting Time



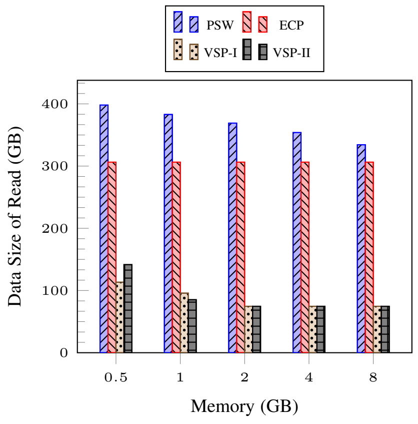
(c) Number of Shards



(d) Number of IOs

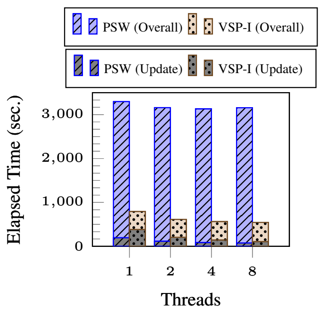


(e) Data Size of Write

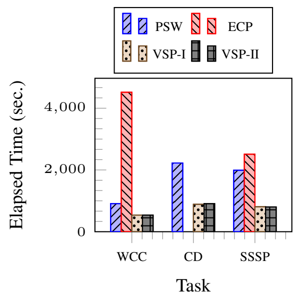


(f) Data Size of Read

Fig. 4. PageRank on the twitter-2010 graph

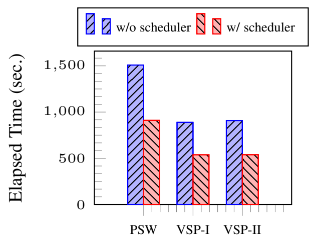


(a) Scaling with Threads

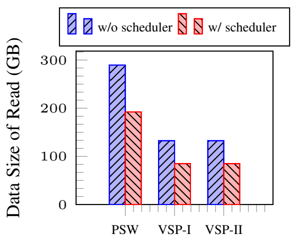


(b) Overall Time of Diffrent Tasks

Fig. 5. Different Tasks on the twitter-2010 graph



(a) Overall Time



(b) Data Size of Read

Fig. 6. WCC on the twitter-2010 graph with or without the scheduler

more threads are added, because the overall time of either PSW or VSP is IO bounded and multi-threading can only reduce the CPU time. If the task is computationally more demanding, it can benefit more from multi-threading. Moreover, while both PSW and VSP are IO bounded, VSP incurs significantly fewer IOs than PSW. As a result, VSP benefits more from the increasing number of threads compared with PSW. In other words, VSP uses multi-core CPUs more effectively.

5.2 Exp-2: More Graph Computing Tasks

Experiment of WCC. After the evaluation under various RAM sizes, we further compare the four algorithms for other graph computing tasks. We set the memory budget as 4 GB for all algorithms. In detail, Figure 5(b) shows the processing time of running WCC over the twitter-2010 graph. The existing algorithms, PSW and ECP, spend 910.0 seconds and 4,516.8 seconds respectively, while our algorithms VSP-I and VSP-II spend 538.2 seconds and 539.2 seconds respectively. In this task, ECP is much slower than others. One reason is that both PSW and our algorithms can employ the scheduler described in Section 4.2 to skip unnecessary updates on some vertices/shards. However, this feature is infeasible for ECP because it is edge-centric and thus cannot support selective scheduling of vertices.

To show the effectiveness of our selective scheduling scheme, we also compare the overall time and the data size accessed by PSW, VSP-I, and VSP-II with or without the scheduler. As shown in Figure 6(a), the scheduler reduces the overall processing

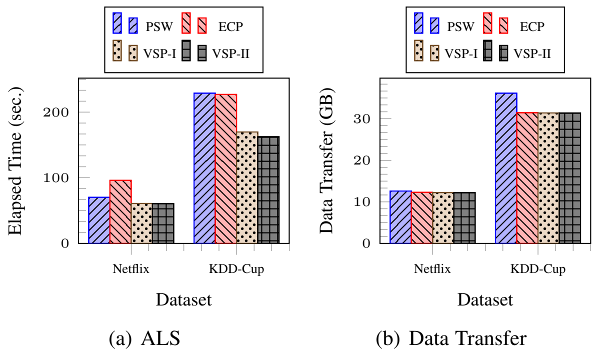


Fig. 7. ALS on the Netflix and KDD-Cup graph

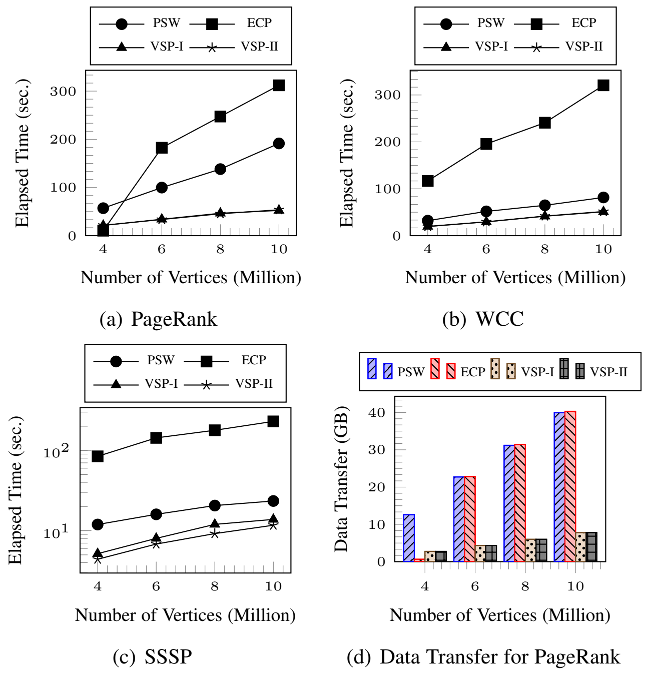


Fig. 8. PageRank, WCC, and SSSP on the synthetic graphs

time for all three algorithms by around 40%. Furthermore, in Figure 6(b), the data size of read for all algorithms is reduced by around 35%.

There is another reason why PSW is faster than ECP. GraphChi employs `zlib` to compress its edge values on disk[4]. In the tasks of WCC and CD, all edge values are integers and can be compressed considerably, which reduces the IO cost dramatically (noting that PSW takes as long as 1,738.9 seconds for WCC without the compression). However, the advantage of compression cannot be found in many other tasks such as PageRank, ALS, and SSSP, where the edge values are real numbers.

Experiment of CD. For the CD task, Figure 5(b) shows the performance of PSW and our algorithms, where our algorithms, VSP-I and VSP-II, clearly outperform PSW. In detail, PSW spends 2,225.0 seconds. VSP-I and VSP-II just take 888.3 seconds and 908.5 seconds respectively. The CD task cannot be accomplished by ECP, because CD is based on label propagation [16], where each vertex chooses the most frequent label among its neighbors in the update function. The most frequent label can be easily decided in terms of vertex-centric processing, where all neighbors and incident edges are passed to the update function. However, this is not the case for the edge-centric processing while ECP cannot iterate all incident edges and all neighbors to complete the required operation.

Experiment of ALS. This task is tested on Netflix and KDD-Cup. The overall processing time is given in Figure 7(a). In this experiment, our algorithms, VSP-I and VSP-II, only outperform other competitors slightly. The reason is that the datasets for this experiment are so small that all four algorithms maintain the values for computation (the latent factors in ALS) in the main memory. Therefore, the total data size being accessed for the four algorithms becomes almost the same as shown in Figure 7(b).

Experiment of SSSP. The result of the last task, SSSP, is shown in Figure 5(b). From the result, we can conclude that our algorithms outperform the other algorithms. Note that this is also another example which shows the benefits of the scheduler in GraphChi and VENUS.

5.3 Exp-3: The Synthetic Graphs

To see how a system performs on graphs with increasing data size, we also did experiments over the 4 synthetic datasets. We test with PageRank, WCC, and SSSP, and report the running time in Figure 8. Again, we see that VSP uses just a fraction of the amount of time as compared to the other two systems. Note that a base-10 log scale is used for the Y axis in Figure 8(c).

In general, the processing time increases with the number of vertices. However, the processing time of PSW and ECP increases much faster than VSP. For example, for the task of PageRank, when the number of vertices increases from 4 million to 10 million, the processing time of PSW increases by 134.4 seconds; and the processing time of ECP increases by 301.6 seconds. In contrast, the processing time of VSP-I and VSP-II just increases by 32.2 seconds and 31.1 seconds respectively. The superior performance of VSP is mainly due to the less amount of data access, as shown in Figure 8(d). There is a special case when the number of vertices is 4 million, in which ECP loads the whole input graph into the memory and outperforms other algorithms.

5.4 Exp-4: The Web-Scale Graph

In this experiment, we compare GraphChi, X-Stream, and VENUS on a very large-scale web graph, clueweb12 [24], which has 978.4 million vertices and 42.5 billion edges. We choose not to use yahoo-web [32] which has been used in many previous works [3], [14], because the density of yahoo-web is incredibly low where 53% of nodes are dangling nodes (nodes with no outgoing edges), and testing algorithms and systems on yahoo-web might give inaccurate speed report. On the other hand, the number of edges in clueweb12 are an order of magnitude bigger and only 9.5% of nodes in clueweb12 are dangling nodes. We run 4 iterations of PageRank for each system. As shown in Table 7, VENUS significantly outperforms GraphChi and X-Stream by reading and writing less amount of data.

5.5 Exp-5: Preprocessing

In this experiment, we evaluate the preprocessing step of GraphChi, X-Stream, and VENUS. GraphChi [3] provides a program called Sharder to preprocess an input graph and divide the

4. This feature was not available in the version reported in [3] and has to be explicitly enabled with all later versions.

TABLE 7
Experiment Results: PageRank on the clueweb12 graph

System	Runtime (hr)	Read (GB)	Write (GB)
PSW	10.8	2939.1	955.6
ECP	16.0	4654.1	2127.4
VSP-I	3.12	977.9	73.0
VSP-II	3.11	1012.2	42.4

graph into multiple shards. Sharder has 4 phases: (1) converting the input graph into a binary adjacency list; (2) counting the in-degree for each vertex and dividing vertices into P intervals; (3) writing each edge to a temporary file of the owning shard; and (4) sorting each shard. X-Stream [14] does not require any partitioning or sorting. They provide `Python` scripts to convert an input graph into a binary edge list. For the sake of fairness, we reimplement these scripts in `C++`.

In Table 6, we present the preprocessing time of GraphChi, X-Stream, and VENUS under the 8 GB memory budget. The preprocessing of VENUS is faster than that of GraphChi, while it is slower than that of X-Stream. This is because X-Stream does not sort the input graphs. However, if we consider both the preprocessing time and the processing time, VENUS is still the fastest one. For example, on the clueweb12 graph, the total execution time of running PageRank on GraphChi, X-Stream, and VENUS are 17.7 hours, 17.0 hours, and 7.4 hours respectively.

Another advantage of our preprocessing method is that we only need to preprocess the input graph once for different memory budgets. In contrast, in order to achieve the best performance, GraphChi must re-shard the input graph if the memory budget changes.

6 Related Systems

There are several options to process big graph tasks: it is possible to create a customized parallel program for each graph algorithm in distributed setting, but this approach is difficult to generalize and the development overhead can be very high. We can also rely on graph libraries with various graph algorithms, but such graph libraries cannot handle web-scale problems [1]. Recently, graph computing over distributed or single machine platform has emerged as a new framework for big data analytics, and it draws intensive interests [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. Broadly speaking, all existing systems can be categorized into the so-called *data-parallel systems* (e.g. MapReduce/Hadoop and extensions) and *graph-parallel systems*.

The data-parallel systems stem from MapReduce. Since MapReduce does not support iterative graph algorithms originally, there are considerable efforts to leverage and improve the MapReduce paradigm, leading to various distributed graph processing systems including PEGASUS [5], GBase [9], Giraph [33], and SGC [34]. On the other hand, the graph-parallel systems use new programming abstractions to compactly formulate iterative graph algorithms, including Pregel [1], Hama [7], Kingeograph [10], Trinity [11], GRACE [19], [20], Horton [35], GraphLab [2], and ParallelGDB [36]. Due to the rapid development of many-core processors, in particular graphical processing units (GPUs), a high performance graph-parallel system that can leverage the computing power of GPUs draws a broad interest [37], [38], [39], [40], [41]. Current efforts focus on efficient high-level abstraction for optimized parallelism and execution over GPUs. There is also work trying to bridge the data-parallel and the graph-parallel systems, such as GraphX [42].

As a recent branch of graph-parallel systems, the disk-based graph computing systems, such as GraphChi [3], X-Stream [14], and TurboGraph [15], have shown great potential in graph analytics, which do not need to divide and distribute the underlying graph over a number of machines, as did in previous graph-parallel systems. And remarkably, they can work with just a single PC on very large-scale problems. It is shown that disk-based graph computing on a single PC can be highly competitive even compared to parallel processing over large scale clusters [3]. A distributed system called MOCgraph also supports out-of-core execution, but there are extensive IOs in managing a huge number of disk-resident messages which significantly affects the overall performance [43].

Disk-based systems. The disk-based systems, including GraphChi [3], TurboGraph [15], and X-Stream [14], are closely related to our work. Both GraphChi and VENUS are vertex-centric. Like our system VENUS, GraphChi also organizes the graph into a number of shards. However, unlike VENUS which requires only a v-shard to be fit into the memory, GraphChi requires each shard to be fit in main memory. As a result, GraphChi usually generates many more shards than VENUS under the same memory constraint (Figure 4(c)), which incurs more data transfer (Figure 4(e) and Figure 4(f)) and random IOs. Furthermore, GraphChi starts the computation after the shard is completely loaded and processes next shard after the value propagation is completely done. In contrast, VENUS enables streamlined processing which performs computation while the data is streaming in. Another key difference of VENUS from GraphChi lies in its use of a fixed buffer to cache the v-shard, which can greatly reduce random IOs.

The TurboGraph system can process graph data without delay, at the cost of limiting its scope on certain embarrassingly parallel algorithms. In contrast, VENUS can deal with almost every algorithms as GraphChi. Different from VENUS that uses hard disk, TurboGraph is built on SSD. X-Stream is edge-centric and allows streamlined processing like VENUS, by storing partial, intermediate results to disk for later access. However, this will double sequential IOs, incur additional computation cost, and increase data loading overhead.

VENUS improves previous systems in several important directions. First, we separate the graph data into the fixed structure table and the mutable value table file, and use a fixed buffer for vertex value access, which almost eliminates the need of batch propagation operation in GraphChi (thus reducing random IOs). Furthermore, each shard in VENUS is not constrained to be fit into memory, but instead, they are concatenated together forming a consecutive file for streamlined processing, which not only removes the batch loading overhead but also enjoys a much faster speed compared to random IOs [14]. Compared to TurboGraph, VENUS can handle a broader set of data mining tasks; compared to X-Stream, VENUS processes the graph data just once (instead of twice in X-Stream) and without the burden of writing the entire graph to disk in the course of computation.

7 Comparison of Computing Models

Various computing abstractions/models for graph computation have been proposed since the seminal work [1]. It may be interesting to have a discussion on their analogy and differences, and expressiveness and limitation. This can help decide the best system and model in implementing a specific algorithm.

- **Vertex-Centric**: This is the seminal model for graph computation proposed by Pregel [1], where a graph algorithm is formulated in terms of a "tiny" vertex update function. The input of an update function for a vertex consists of data on the adjacent vertices and edges. A vertex can also exchange messages with any other vertices. GraphLab [2] and GraphChi [3] use a similar model, but they only allow vertices to exchange messages with their adjacent vertices. MOCgraph [43] tries to improve the memory utilization of Giraph [33] (an open-source implementation of Pregel) by processing messages online, which requires the messages to be commutative and thus restricts its expressiveness [43].
- **Gather-Apply-Scatter (GAS)**: This computing model is made popular by PowerGraph [44], which further refines the update function into 3 sub-functions: gather, apply, and scatter. This abstraction enables better graph partition for power-law graphs in a distributed setting [44]. However, PowerGraph uses a user-defined `sum` operation to combine messages, which must be commutative and associative like a numerical sum. This makes the GAS model less expressive than the vertex-centric model. PowerGraph can emulate the vertex-centric model, but this will eliminate the benefits of the GAS model [44].
- **Edge-Centric**: X-Stream [14] proposes an edge-centric scatter-gather model, which can be seen as a variant of the GAS model. In this model, there are only two user-defined functions, scatter and gather, which are executed on edges instead of vertices. This model is even more restrictive than the GAS model, e.g., it is unclear how to implement WCC under this model as explained in Section 5.2.
- **VSP**: The proposed VSP model requires data on each edge to be read-only. For most graph algorithms, the mutable value on edge (v, w) can be computed based on the mutable value of vertex v and the read-only attribute on edge (v, w). As such, we can represent all mutable values of the out-edges of vertex v by a fixed-length mutable value on vertex v. A similar design is also used in recent papers [19], [20]. This constraint does prevent some algorithms from being implemented in VENUS without modifications. One example we found is belief propagation [45] in which the mutable value on edge (v, w) must be computed based on the mutable value of vertex v and itself in the previous iteration recursively. To support algorithms like belief propagation, we could extend VENUS by storing mutable edge values in the value table, but this may eliminate the benefits of the VSP model. We conclude that the VSP model is general enough to support most graph algorithms including many popular random walk models such as [46], [47].

In some scenarios, graphs may evolve over time. While we focus mostly on static graphs in this paper, VENUS can also support evolving graphs with some simple extensions. As discussed in Section 4, the structure table stores graph edges sorted by destination and is physically organized into disk blocks. This can be easily extended with an in-memory buffer corresponding to each disk block. In handling changes over the existing graph, new incoming edges can be maintained in the corresponding in-memory buffer. Then, the operation of scanning the structure table also needs to include retrieving edges from the in-memory buffer. An in-memory buffer will be written back to disk when it is full, followed by necessary block splitting. Similar ideas can also be applied for adding new vertices and v-shards only need to be updated when a certain number of new vertices have been added.

8 Conclusions

We have presented VENUS, a disk-based graph computation system that is able to handle billion-scale problems efficiently on just a single commodity PC. It includes a novel design for graph storage, a new data caching strategy, and a new external graph computing model that implements vertex-centric streamlined processing. In effect, it can significantly reduce data access, minimize random IOs, and effectively exploit main memory. Extensive experiments on 4 large-scale real-world graphs and 4 large-scale synthetic graphs show that VENUS can be much faster than GraphChi and X-Stream, two state-of-the-art disk-based systems.

Acknowledgments

The work is partly supported by NSFC of China (Grant No. 61103049).

References

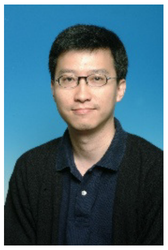
[1] G. Malewicz, M. Austern, and A. Bik, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–145.
[2] Y. Low, D. Bickson, and J. Gonzalez, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.
[3] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *OSDI*, 2012, pp. 31–46.
[4] X. Martinez-Palau and D. Dominguez-Sal, "Analysis of partitioning strategies for graph processing in bulk synchronous parallel models," in *CloudDB*, 2013, pp. 19–26.
[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *ICDM*. IEEE, 2009, pp. 229–238.
[6] R. Chen, X. Weng, B. He, and M. Yang, "Large Graph Processing in the Cloud," in *SIGMOD*, 2010, pp. 1123–1126.
[7] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An Efficient Matrix Computation with the MapReduce Framework," in *CLOUDCOM*, 2010, pp. 721–726.
[8] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal, "HipG: Parallel Processing of Large-Scale Graphs," *SIGOPS Operating Systems Review*, vol. 45, no. 2, pp. 3–13, 2011.
[9] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "GBASE : A Scalable and General Graph Management System," in *KDD*, 2011, pp. 1091–1099.
[10] R. Cheng, F. Yang, and E. Chen, "Kineograph : Taking the Pulse of a Fast-Changing and Connected World," in *EuroSys*, 2012, pp. 85–98.
[11] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *SIGMOD*, 2013.
[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, 2004.
[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," in *HotCloud*, 2010.
[14] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric Graph Processing using Streaming Partitions," in *SOSP*, 2013.
[15] W. Han, S. Lee, K. Park, and J. Lee, "TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC," in *KDD*, 2013, pp. 77–85.
[16] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Carnegie Mellon University, Tech. Rep., 2002.
[17] 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*, 2015, pp. 1131–1142.
[18] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *KDD*, 2012.
[19] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous Large-Scale Graph Processing Made Easy," in *CIDR*, 2013.
[20] W. Xie, G. Wang, and D. Bindel, "Fast iterative graph computation with block updates," *PVLDB*, pp. 2014–2025, 2013.

[21] A. Aggarwal and J. S. Vlller, "The input/output complexity of sorting and related problems," *CACM*, vol. 31, no. 9, pp. 1116–1127, 1988.
[22] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008.
[23] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter , a Social Network or a News Media?" in *WWW*, 2010.
[24] "Clueweb12 web graph," http://www.lemurproject.org/clueweb12/index.php, 2013.
[25] "Laboratory for web algorithmics - datasets," http://law.di.unimi.it/datasets.php.
[26] P. Boldi and S. Vigna, "The WebGraph Framework I : Compression Techniques," in *WWW*, 2004, pp. 595–602.
[27] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *WWW*, 2011, pp. 1–13.
[28] J. Bennett and S. Lanning, "The netflix prize," in *KDD-Cup Workshop*, 2007.
[29] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11." *JMLR W&CP*, pp. 3–18, 2012.
[30] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Tech. Rep., 1999.
[31] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale Parallel Collaborative Filtering for the Netflix Prize," in *AAIM*, 2008, pp. 337–348.
[32] "Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, circa 2002," http://webscope.sandbox.yahoo.com/.
[33] "Giraph," http://giraph.apache.org/.
[34] L. Qin, J. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable Big Graph Processing in MapReduce," in *SIGMOD*, 2014.
[35] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel, "Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs," *PVLDB*, vol. 6, no. 14, pp. 1918–1929, 2013.
[36] L. Barguñó, D. Dominguez-sal, V. Muntés-mulero, and P. Valduriez, "ParallelGDB: A Parallel Graph Database based on Cache Specialization," in *IDEAS*, 2011, pp. 162–169.
[37] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *TPDS*, vol. 25, no. 6, pp. 1543–1552, 2013.
[38] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-Centric Graph Processing on GPUs Categories and Subject Descriptors," in *HPDC*, 2014, pp. 239–251.
[39] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs," in *GRADES*, 2014.
[40] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU," in *PPoPP Poster*, 2015, pp. 265–266.
[41] S. Che, "GasCL: A Vertex-Centric Graph Model for GPUs," in *HPEC*, 2014, pp. 1–6.
[42] 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*, 2014.
[43] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "MOCgraph : Scalable Distributed Graph Processing Using Message Online Computing," *PVLDB*, vol. 8, no. 4, pp. 377–388, 2014.
[44] J. E. Gonzalez, D. Bickson, and C. Guestrin, "PowerGraph : Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, 2012, pp. 17–30.
[45] J. Pearl, "Reverend Bayes on inference engines: A distributed hierarchical approach," in *AAAI*, 1982, pp. 133–136.
[46] X.-M. Wu, Z. Li, A. M.-C. So, J. Wright, and S.-F. Chang, "Learning with partially absorbing random walks," in *NIPS*, 2012, pp. 1–9.
[47] Z. Li, Y. Fang, Q. Liu, J. Cheng, R. Cheng, and J. C. Lui, "Walking in the Cloud: Parallel SimRank at Scale," *PVLDB*, vol. 9, no. 1, 2015.

Qin Liu is a PhD candidate in the Department of Computer Science & Engineering at The Chinese University of Hong Kong. He received his Bachelor degree in Computer Science from Shanghai Jiao Tong University. His research interests are mainly in computer systems, graph mining, and algorithm design.

Jiefeng Cheng is a researcher with Huawei Noah's Ark Laboratory, Hong Kong. He obatained his PhD from the Chinese University of Hong Kong in 2007. From 2007 to 2010, he worked as a research fellow at the Chinese University of Hong Kong and the Hong Kong University, respectively. Prior to joining Huawei, he was a faculty memeber at the University of Chinese Academy of Sciences, China. His research interests include graph mining and management.

Zhenguo Li is currently a researcher in Huawei Noah's Ark Lab at Hong Kong. He received the B.S. and M.S. degrees from the Department of Mathematics at Peking University, in 2002 and 2005, respectively, and the Ph.D. degree from the Department of Information Engineering at the Chinese University of Hong Kong, in 2008. He was an associate research scientist in the Department of Electrical Engineering at Columbia University. His research interests include machine learning and artificial intelligence.



John C.S. Lui is currently a full professor in the Department of Computer Science & Engineering at The Chinese University of Hong Kong. He received his Ph.D. in Computer Science from UCLA. After his graduation, he joined the IBM Almaden Research Laboratory/San Jose Laboratory and participated in various research and development projects on file systems and parallel I/O architectures. His current research interests are in Internet, network sciences with large data implications (e.g., online social networks), machine learning on large data analytics, network/system security (e.g., cloud security, mobile security), network economics, cloud computing, large scale distributed systems and performance evaluation theory. John has been serving in the editorial board of IEEE/ACM Transactions on Networking, IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Journal of Performance Evaluation, Journal of Network Science and International Journal of Network Security. John received various departmental teaching awards and the CUHK Vice-Chancellor's Exemplary Teaching Award. John also received the CUHK Faculty of Engineering Research Excellence Award (2011-2012). John is a co-recipient of the best paper award in the IFIP WG 7.3 Performance 2005, IEEE/IFIP NOMS 2006, and SIMPLEX 2013. He is an elected member of the IFIP WG 7.3, Fellow of ACM, Fellow of IEEE, Senior Research Fellow of the Croucher Foundation and is currently the chair of the ACM SIGMETRICS.