VENUS: Vertex-Centric Streamlined Graph Computation on a Single PC

Jiefeng Cheng[1], Qin Liu[2], Zhenguo Li[1], Wei Fan[1], John C.S. Lui[2], Cheng He[1]

[1]*Huawei Noah's Ark Lab, Hong Kong*
{cheng.jiefeng,li.zhenguo,david.fanwei,hecheng}@huawei.com
[2]*The Chinese University of Hong Kong*
{qliu,cslui}@cse.cuhk.edu.hk

***Abstract*—Recent studies show that disk-based graph computation on just a single PC can be as highly competitive as cluster-based computing 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 problems efficiently on a commodity PC. VENUS adopts a novel computing architecture that features vertex-centric "streamlined" processing – the graph is sequentially loaded and the update functions are executed in parallel on the fly. VENUS deliberately avoids loading batch edge data by separating read-only structure data from mutable vertex data on disk. Furthermore, it minimizes random IOs by caching vertex data in main memory. The streamlined processing is realized with efficient sequential scan over massive structure data and fast feeding a large number of update functions. Extensive evaluation on large real-world and synthetic graphs has demonstrated the efficiency of VENUS. For example, VENUS takes just 8 minutes with hard disk for PageRank on the Twitter graph with 1.5 billion edges. In contrast, Spark takes 8.1 minutes with 50 machines and 100 CPUs, and GraphChi takes 13 minutes using fast SSD drive.**

I. 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 on October 4, 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 processing these big datasets, due to the difficulties in managing these data with our current methodologies or data mining software tools.

Graph computing over distributed or single 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 computation model, in which the user just needs to provide a simple update function to the system that is executed for each vertex in parallel [1], [2], [3]. These developments substantially advance our ability to analyze large-scale graph data, which cannot be efficiently handled by previous parallel abstractions such as MapReduce [12] due to 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, running the PageRank algorithm on a Twitter graph of 1.5 billion edges, Spark needs 8.1 minutes with 50 machines (100 CPUs) [16], while GraphChi only spends 13 minutes with just a single MacMini of 8GB RAM and 256GB SSD drive; for the belief propagation algorithm on a Web graph of 6.7 billion edges, PEGASUS takes 22 minutes with 100 machines [5], while GraphChi uses 27 minutes on a single PC. These results suggest that disk-based graph computing 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 a large number of update functions. 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. 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 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 become a bottleneck of the disk-based approach. This generates great interests in new architectures for efficient disk-based graph computation.

The seminal work for disk-based graph computation is the GraphChi system [3]. GraphChi organizes a graph into a number of shards and processes each shard 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 TurboGraph system [15] addresses these issues with a new computing model, *pin-and-sliding* (PAS), and by using expensive SSD. PAS aims to process incoming, partial graph data without delay, but it is applicable only to certain embarrassingly parallel algorithms. 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 data loading overhead. Since the ECP model uses very different APIs from previous vertex-centric based system, the user need to re-implement many graph algorithms on ECP which causes high development overhead. Moreover, certain important graph algorithms such as community detection [17] cannot be implemented on the ECP model (explained in Section IV-B).

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 vertex-centric computation 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 as much as possible and parallelize computation and disk IO at the same time. 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 less shards than the existing scheme [3], and effectively utilizes large 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 different 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 computation model and other models, by measuring the amount of data transferred between disk and 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 available memory increases, till an in-memory model is emerged where the least overhead is achieved; in contrast, existing approaches just switch between the in-memory model and the disk-based model, where the performance can be radically different. The purpose of this new analysis method is to clarify the essential factors for good performance instead of a thorough comparison of different systems. Moreover, it opens a new way to evaluate disk-based systems theoretically.

Extensive experiments. We did extensive experiments using both large-scale real-world graphs and large-scale synthetic graphs to validate the performance of our approach. Our experiments look into the key performance factors to all disk-based systems including computational time, the effectiveness of main memory utilization, the amount of data read and write, and the number of shards. And we found that VENUS is up to 3x faster than GraphChi and X-Stream, two state-of-the-art disk-based systems.

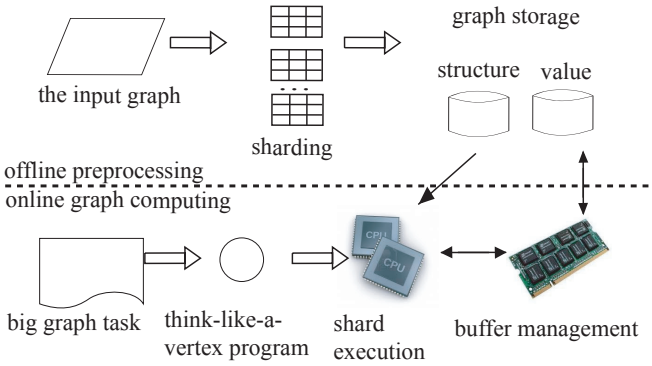


Fig. 1. The VENUS Architecture

The rest of the paper is organized as follows. Section II gives an overview of our VENUS system, which includes a disk-based architecture, graph organization and storage, and an external computing model. Section III presents algorithms to substantialize our processing pipeline. We extensively evaluate VENUS in Section IV. Section V reviews more related work. Section VI concludes the paper.

II. System Overview

VENUS is based on a new disk-based graph computing architecture, which supports a novel graph computing model: *vertex-centric streamlined processing* (VSP) such that the graph is sequentially loaded and the update functions are executed 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 and with CPU, memory and disk access. By working together, the system significantly reduces the amount of data to be accessed, generates a smaller number of shards than the existing scheme [3], and effectively utilizes large main memory with provable performance guarantee.

A. 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 [18], [19], the user can assign a mutable vertex value to each vertex and define an arbitrary read-only edge value on each edge. Note that this does not result in any loss of expressiveness, since mutable data associated with edge (u, v) can be stored in vertex u. Let (u, v) be a directed edge from node u to node v. Node 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,

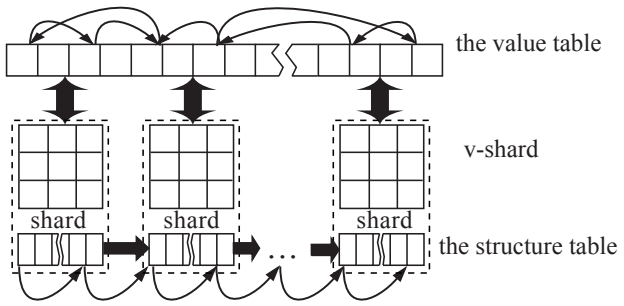


Fig. 2. Vertex-Centric Streamlined Processing

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 disk-based approach. Therefore, a disk-based graph computing system needs to manage the storage and the use of memory and CPU in an intelligent way to minimize the amount of disk data to be accessed.

VENUS, its architecture depicted in Fig. 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: (1) offline preprocessing and (2) online computing. The offline preprocessing constructs the graph storage, while the online computing manages the interaction between CPU, memory, and disk.

B. 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 update functions. 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 II-C. 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

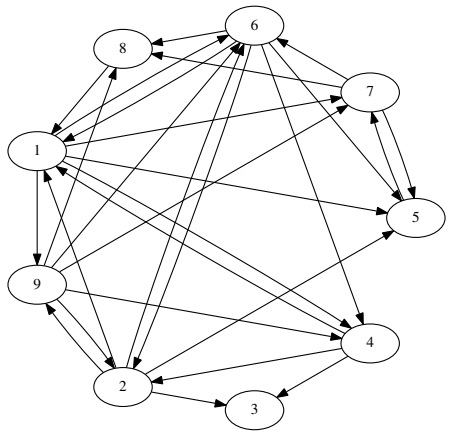


Fig. 3. Example Graph

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 as shown in Fig. 3. Suppose the vertices are divided into three intervals: $I_1 = [1, 3]$, $I_2 = [4, 6]$, and $I_3 = [7, 9]$. Then, the resulting shards, including g-shards and v-shards, are listed in Table I.

In practice, all g-shards are further concatenated to form the structure table, i.e., a stream of structure records (Fig. 2). 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 as few number of shards 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 v-shard is loaded and then held in the main memory, VENUS can readily execute the update functions of all vertex with only "one sequential scan" over the structure table of the g-shard. We will discuss how to load and update vertex values for vertices in each v-shard in Section III.

Graph storage. We propose a new graph storage that aims to reduce the amount of data to be accessed. 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 need to be scanned once and only once, while the value data usually need to be accessed multiple times as 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 is 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 into main memory as compared with the existing

TABLE I
SHARDING EXAMPLE: VENUS

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

TABLE II
SHARDING EXAMPLE: GRAPHCHI

Interval	$I_1 = [1, 2]$	$I_2 = [3, 5]$	$I_3 = [6, 7]$	$I_4 = [8, 9]$
Shard	2 → 1	1 → 4,5	1 → 6,7	1 → 9
	4 → 1,2	2 → 3,5	2 → 6	2 → 9
	6 → 1,2	4 → 3	5 → 7	6 → 8
	8 → 1	6 → 4,5	7 → 6	7 → 8
	9 → 2	7 → 5	5 → 6,7	9 → 8
		9 → 4		

system GraphChi [3] and instead caches value data in 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 the $|V|$ 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). Our key observation is that, for most graph algorithms, the mutable value on a directed edge (u, v) can be computed based on the mutable value of vertex u and the read-only attribute on edge (u, v). Consequently, we can represent all mutable values of the out-edges of vertex u by a fixed-length mutable value on vertex u.

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 immediately 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 Fig. 2.

We use the graph in Fig. 3 to illustrate and compare the processing pipelines of VENUS and GraphChi. The sharding structures of VENUS are shown in Table I, and those for GraphChi are in Table II 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 the in-edges of vertex 1 (which include $(2, 1)$, $(4, 1)$, $(6, 1)$, and $(8, 1)$), we can perform the value update on vertex 1, and at the same time, we load the in-edges of vertices 2 and 3 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 shards. More critically, because all in-edge and out-edges must be loaded before computation starts, GraphChi cannot parallelize IO operations and computations like VENUS.

TABLE III
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 within a structure record
B	size of a disk block that requires unit IO to access it

C. 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 main memory per iteration; 2) the number of shards; and 3) 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 or solid-state disk (SSD), and various types of memory hierarchy including on-board cache/RAM and RAM/disk; (ii) it can be used to derive IO complexity as in Section III-C, which can be based on accessing that certain amount of data with block device; and (iii) it helps us to examine other criteria, including the number of shards and large memory adaption. We summarize the results in Table IV, 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.

For easy reference, we list the notation in Table III. For our VSP model, V is split into P disjoint intervals. Each interval I has 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 $S(I) = \{u \notin I | (u, v) \in gs(I)\}$. We have

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

Let δ be a scalar such that

$$\sum_I |S(I)| = \delta m, \text{ where } 0 \leq \delta \leq 1.$$

It can be seen that

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

Let C be the size of a vertex value record, and let D be the size of one edge field within a structure record. We use B to denote the size of a disk block that requires unit IO to access it. According to [14], B equals to 16MB for hard disk and 1MB for SSD.

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. When 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 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 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 update each vertex, so the size of data 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 III, 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 needed to 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 $vs(I_i) \leq M$ 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 $M' \geq |vs(I_1)| + |vs(I_2)| \geq M$. Under the memory size M', we can merge interval I_1 and I_2 into I_t, because $|vs(I_t)| \leq |vs(I_1)| + |vs(I_2)| \leq M'$. 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 $|S(I_t)| \leq |S(I_1)| + |S(I_2)|$. Therefore we have $\delta' \leq \delta$, which means as M increases, δ becomes smaller. When $M \geq Cn$, we have $P = 1$ where $\delta = 0$. In such a single shard case, the data size of read reaches the lower bound $Cn + Dm$.

III. Algorithm

In this section, we discuss the full embodiment of our vertex-centric streamlined processing model, to describe 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 flavor and IO complexity in implementing the processing of Section II. Note that the IO results here are consistent with the data transfer size results in Section II 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 computing. 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 to P, namely 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 to 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.

A. 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

TABLE IV
COMPARISON OF DATA TRANSFERRED BETWEEN SINGLE-PC GRAPH COMPUTING SYSTEMS

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

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 $\leftarrow$ `UpdateVertex(`$R(v)$, VB`)`;
7 **return**;

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`)`
11 **return**;

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 to 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 multiple disk pages for vertices of large degrees. Hence, there are $|V|$ such records in total. As an example, for the graph in Fig. 3, the structure record $R(4)$ of vertex 4 contains incoming vertices 1, 6, and 9 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 III-B. 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 Fig. 3 and its sharding structures in Table I. Suppose $I = I_1$. For the value buffer VB, the frame table contains value table pages of vertex 1, 2, and 3 in I_1, and the map contains vertex values of vertex 4, 6, 8, and 9 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. In Procedure ExecuteVertex, note that 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.

B. Two Algorithms

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 II-B, we split V into P disjoint intervals, such that the vertex values of each v-shard can be 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 consecutively. And more importantly, the value records in the $(j+1)$-th interval are stored immediately after the value records for the j-th. Therefore, similar to the parallel sliding window of [3], when the active shard is shift 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 Fig. 3 and Table I. 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 I).

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 ∉ 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), VB, I)
9 return;
```

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 Fig. 3 and Table I. Suppose that we want to update interval I_1. First, we need to load $S(I_1) = \{4, 6, 8, 9\}$ 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.

C. Analysis of IO costs

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 [20]. In this model, the IO cost of an algorithm is the number of block transfers from disk to main memory adding the number of non-sequential seeks. So the complexity is parametrized by the size of block transfer, B.

For Algorithm 1, the size of data read is $C(n+\delta m)+Dm$ obtained from Table IV. 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

TABLE V
BIG-O BOUNDS IN THE IO MODEL OF SINGLE-MACHINE GRAPH PROCESSING SYSTEMS

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	$P\frac{Cn}{B} + \frac{Dm}{B}$	$\frac{Cn}{B}$

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 V 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.

IV. PERFORMANCE

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. VENUS is implemented in C++. We ran each experiment three times and reported the averaged execution time. We deliberately avoid caching disk pages by the operating system as explained in Section IV-A. All algorithms are evaluated using hard disk, so we do not include TurboGraph [15] due to its requirement of SSD drive on the computer. Like GraphChi and X-Stream, VENUS allows user to explicitly specify a main memory budget. Specifically, we spend half of the main memory budget for the frame table in VB, which are 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 rest memory for storing auxiliary data. All experiments are conducted on a commodity machine with Intel i7 quad-core 3.4GHz CPU, 16GB RAM, and 4TB hard disk, running Linux.

We mainly examine four important aspects of a system which are key to its performance: 1) computational time; 2) the effectiveness of main memory utilization; 3) the amount of data read and write; and 4) the number of shards. We experiment over 4 large real-world graph datasets, Twitter [21], clueweb12 [22], Netflix [23], and Yahoo! Music user ratings used in KDD-Cup 2011 [24] as well as synthetic graphs. We use the SNAP graph generator[1] 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 VI. We consider the following data mining tasks, 1) PageRank [25]; 2) Weakly Connected Components (WCC) [17]; 3) Community Detection (CD) [17]; and 4) Alternating Least Squares (ALS) [26]. Our four algorithms essentially represent graph applications in the categories of iterative matrix operations (PageRank), graph mining (WCC and CD), and collaborative filtering (ALS). Certain graph algorithms like belief propagation [3], [14] that require the mutable value of one edge to be computed recursively based on the mutable values of other edges, cannot be implemented on VENUS without modifications.

TABLE VI
FOUR REAL-WORLD GRAPH DATASETS AND FOUR SYNTHETIC GRAPH DATASETS USED IN OUR EXPERIMENTS

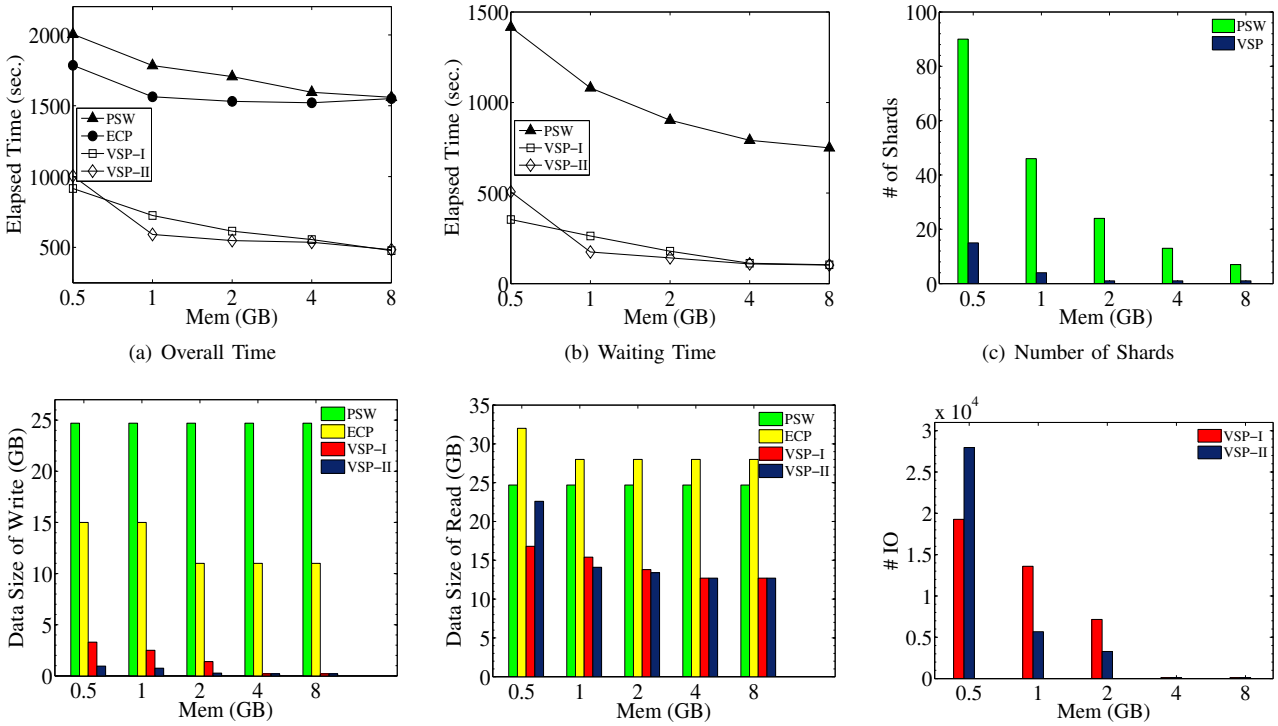
Dataset	# Vertex	# Edge	Type
Twitter	41.7 million	1.4 billion	Directed
clueweb12	978.4 million	42.5 billion	Directed
Netflix	0.5 million	99.0 million	Bipartite
KDDCup	1.0 million	252.8 million	Bipartite
Synthetic-4m	4 million	54.37 million	Directed
Synthetic-6m	6 million	86.04 million	Directed
Synthetic-8m	8 million	118.58 million	Directed
Synthetic-10m	10 million	151.99 million	Directed

In Table VII, we report the preprocessing time of GraphChi and VENUS under 8GB memory budget. The preprocessing step of our system is split into 3 phases: (1) counting degree for each vertex (requiring one pass over the input file) and dividing vertices into P intervals; (2) writing each edge to a temporary scratch file of the owning interval; and (3) sorting edges by their source vertices in each scratch file to construct a v-shard and g-shard file; in constructing the v-shard and g-shard file in (3), the processing consists of merging adjacent intervals and counting distinct vertices in those corresponding scratch files till the memory budget is reached. These phases are identical to those used in GraphChi except that we further merge adjacent intervals in phase 3. The total IO cost of preprocessing is $5\frac{|E|}{B} + \frac{|V|}{B}$ (B is block size) which is the same as GraphChi [3]. Therefore, the preprocessing cost is proportional to the graph size. These results are verified in Table VII. Note that there is no preprocessing in X-Stream.

A. Exp-1: PageRank on Twitter Graph

The first experiment runs PageRank on the Twitter graph. We compare the four algorithms (PSW, ECP, VSP-I, VSP-II) under various memory budgets from 0.5GB to 8GB. Note that PSW and VSP-I/VSP-II use Linux system calls to access data from disk, where the operating system caches data in its *pagecache*. This allows PSW and VSP-I/VSP-II 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

[1] http://github.com/snap-stanford/snap



(a) Overall Time (b) Waiting Time (c) Number of Shards

(d) Data Size of Write (e) Data Size of Read (f) Number of IOs

Fig. 4. PageRank on Twitter Graph

TABLE VII
PREPROCESSING TIME (SEC.) OF GRAPHCHI AND VENUS

Dataset	GraphChi	VENUS
Twitter	424	570
clueweb12	19,313	18,845
Netflix	180	75
KDDCup	454	169
Synthetic-4M	17	18
Synthetic-6M	23	30
Synthetic-8M	34	41
Synthetic-10M	47	53

use pagecache-mangagement[2] to disable pagecache in all our experiments.

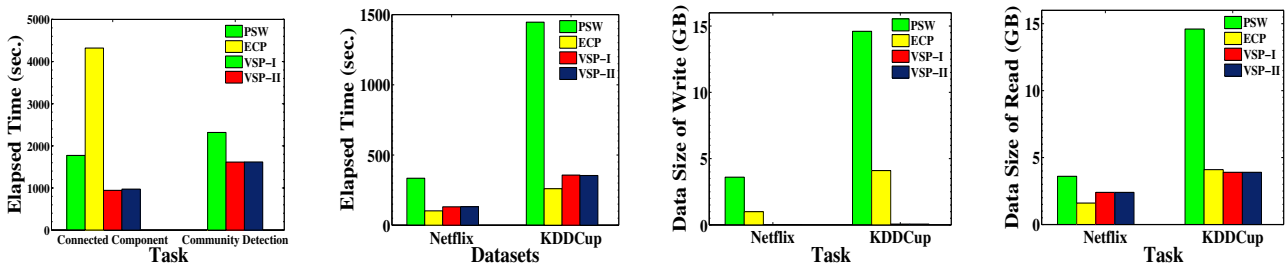
The results of processing time are reported in Fig. 4(a), where we can see that VSP is up to 3x faster than PSW and ECP. For example, in the case that the budget of main memory is 8GB, PSW spends 1559.21 seconds. ECP also needs 1550.2 seconds. However, VSP-I and VSP-II just need 477.39 and 483.47 seconds, 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.

In Fig. 4(b), we compare PSW and VSP in terms of the overall waiting time before executing a next shard. 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 budget of main memory is 8GB, PSW spends 749.78 seconds. However, VSP-I and VSP-II just needs 104.01 and 102.12 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.

VSP also generates significantly smaller number of shards than PSW, as shown in Fig. 4(c). For example, in the case that the budget of main memory is 0.5GB and 1GB, PSW generates 90 and 46 number of shards, respectively. And these numbers for our algorithms are 15 and 4. This is because VSP spends the main budget of the memory on the value data of a v-shard, while the space needed to keep related structure data in memory is minimized.

Fig. 4(d) and Fig. 4(e) 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, PSW has to write 24.7GB data to disk, and read the same amount of data from disk, per iteration, regardless of memory size. These numbers for ECP are 11GB and 28GB under 8GB memory budget, which are also very large and become a significant setback of ECP in its edge-centric streamlined processing. In sharp contrast, VSP only writes 0.24GB, which is 100X and 50X smaller than PSW and ECP, respectively. In terms of data size of read, VSP reads 12.7-16.8GB data under various memory budgets. The superiority of

[2]https://code.google.com/p/pagecache-mangagement/



(a) WCC & CD (b) ALS (c) Data Size of Write (d) Data Size of Read

Fig. 5. More Graph Mining Tasks

VSP in data access is mainly due to the separation of structure data and value data and caching the value data in a fixed buffer.

Although VSP-I and VSP-II perform very closely, they have slight difference in terms of IO performance, as shown in Fig. 4(f). For most cases, VSP-II incurs less IOs than VSP-I because it is free of maintaining the materialized view in executing shards. However, when the memory budget is smaller than 1GB, the number of P increases quickly. In this case, VSP-II is slower due to the heavy access of the value table.

B. Exp-2: More Graph Mining Tasks

After the evaluation under various RAM sizes, we further compare the four algorithms for other graph mining tasks. We set the memory budget as 4GB for all algorithms. In detail, Fig. 5(a) shows the processing time of running WCC and CD over Twitter, where our algorithms, VSP-I and VSP-II, clearly outperform the other competitors. For example, in terms of the WCC task, the existing algorithms, PSW and ECP, spend 1772.57 and 4321.06 seconds, respectively, while our algorithms spend 942.74 and 972.43, respectively. In this task, ECP is much slower than PSW. One reason is that both PSW and our algorithms can employ the *selective scheduling* [3], [19] 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.

For the CD task, Fig. 5(a) shows the performance of PSW and our algorithms. In detail, PSW spends 2317.75 seconds. VSP-I and VSP-II just need 1614.65 and 1617.04 seconds, respectively. The CD task cannot be accomplished by ECP, because CD is based on label propagation [17], 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.

The next task is ALS, which is tested over both datasets of Netflix and KDDCup. The overall processing time is given in Fig. 5(b). In this test, the performance of PSW is much slower than ECP and our algorithms, but ECP is slightly better than both VSP-I and VSP-II. For example, in terms of the ALS task over KDDCup, PSW spends 1446.01 seconds. ECP spends 259.99 seconds. VSP-I and VSP-II spend 357.04 and 190.32 seconds, respectively.

We compare the total data size being accessed per iteration of the four algorithms in Fig. 5(c) and Fig. 5(d). ECP still accesses more data than we do. For example, ECP has to access 2.64GB and 8.23GB disk data, including both read and write, for Netflix and KDDCup, respectively. For VSP-I and VSP-II, these numbers are just 2.41 and 3.96. However, our VSP is slightly slower, because VSP requires more non-sequential seeks than ECP. Finally, note that because VSP-I and VSP-II are both working in the in-memory mode due to the small graph size of Netflix and KDDCup, so they read/write the same amount of data size.

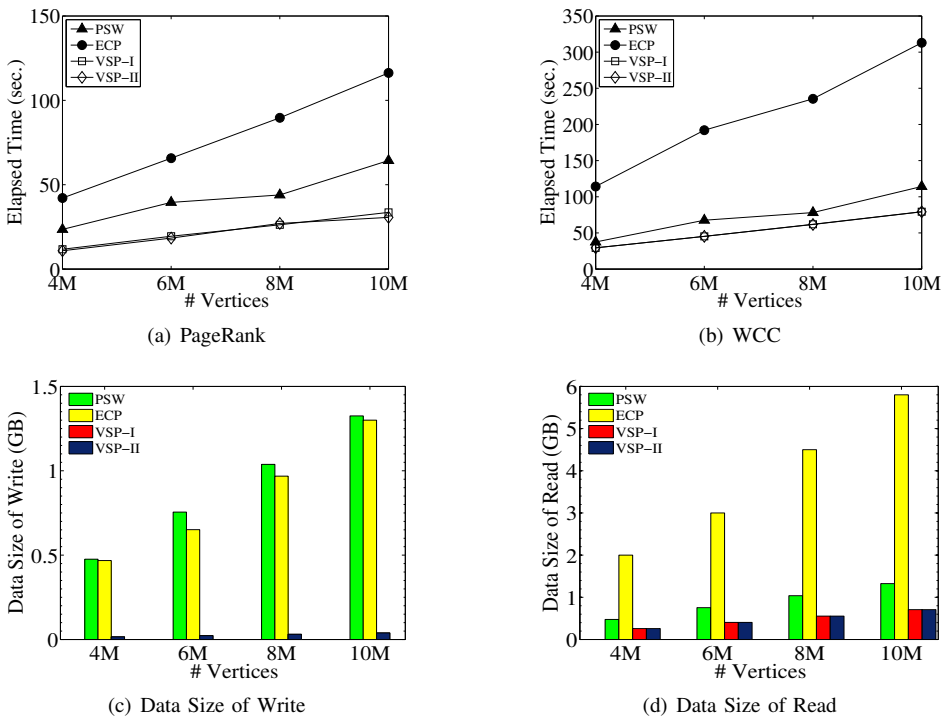
C. Exp-3: The Synthetic Datasets

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 and WCC, and report the running time in Fig. 6(a) and Fig. 6(b) respectively. Again, we see that VSP uses just a fraction of the amount of time as compared to the other two systems.

In general, the processing time increases with the number of vertices. However, the time of PSW and ECP increases much faster than VSP. For example, when the number of vertices increases from 4 million to 10 million, the time of PSW increases by 40.81 and 76.68 seconds for the task of PageRank and WCC, respectively; and the time of ECP increases by 74.13 and 198.72 seconds. In contrast, the time of VSP-1 just increases by 21.85 and 49.93 seconds. The superior performance of VSP is mainly due to the less amount of data access per iteration, as shown in Fig. 6(c) and Fig. 6(d).

D. Exp-4: On the Web-Scale Graph

In this experiment, we compare GraphChi, X-Stream, and VENUS on a very large-scale web graph, clueweb12 [22], which has 978.4 million vertices and 42.5 billion edges. We choose not to use yahoo-web [27] 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 2 iterations of PageRank for each system. As shown in Table VIII, VENUS significantly outperforms GraphChi and X-Stream by reading and writing less amount of data.



(a) PageRank (b) WCC

(c) Data Size of Write (d) Data Size of Read

Fig. 6. The Synthetic Datasets

TABLE VIII
EXPERIMENT RESULTS: PAGERANK ON CLUEWEB12

System	Time	Read	Write
PSW	15,495 s	661GB	661GB
ECP	26,702 s	1,121GB	571GB
VSP-I	7,074 s	213GB	43GB
VSP-II	6,465 s	507GB	19GB

V. 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 multi-core 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 [28], and SGC [29]. 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], [18], Horton [30], GraphLab [2], and ParallelGDB [31]. There is also work trying to bridge the two categories of systems, such as GraphX [32]. 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].

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 (Fig. 4(c)), which incurs more data transfer (Fig. 4(d) and Fig. 4(e)) 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 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.

VI. Conclusion

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. In future work, we plan to improve our selective scheduling of vertex updates and extend our system to SSD, which will further accelerate VENUS greatly.

Acknowledgments

The work is partly supported by NSFC of China (Grant No. 61103049) and 973 Fundamental R&D Program (Grant No.2014CB340304). The authors would like to thank the anonymous reviewers for their helpful comments.

References

[1] G. Malewicz, M. Austern, and A. Bik, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=1807184

[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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2212354

[3] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi : Large-Scale Graph Computation on Just a PC Disk-based Graph Computation," 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, Dec. 2009, pp. 229–238.

[6] R. Chen, X. Weng, B. He, and M. Yang, "Large Graph Processing in the Cloud," pp. 1123–1126, 2010.

[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*. Ieee, Nov. 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. [Online]. Available: http://research.microsoft.com/jump/183710

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, vol. 51, no. 1. ACM, 2004, pp. 107–113.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," in *HotCloud*, 2010, pp. 10–10.

[14] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric Graph Processing using Streaming Partitions," in *SOSP*, 2013, pp. 472–488.

[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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487581

[16] C. Engle, A. Lupher, and R. Xin, "Shark: fast data analysis using coarse-grained distributed memory," in *SIGMOD Demo*, 2012, pp. 1–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2213934

[17] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Tech. Rep., 2002. [Online]. Available: http://lvk.cs.msu.su/~bruzz/articles/classification/zhu02learning.pdf

[18] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous Large-Scale Graph Processing Made Easy," in *CIDR*, 2013.

[19] W. Xie, G. Wang, and D. Bindel, "Fast iterative graph computation with block updates," *PVLDB*, pp. 2014–2025, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2556581

[20] A. Aggarwal and J. S. Vlller, "The input/output complexity of sorting and related problems," *CACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[21] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *WWW*, 2010, pp. 591–600.

[22] Evgeniy Gabrilovich, M. Ringgaard, and A. Subramanya, "FACC1: Freebase annotation of ClueWeb corpora, Version 1 (Release date 2013-06-26, Format version 1, Correction level 0)"," http://lemurproject.org/clueweb12/, 2013.

[23] J. Bennett and S. Lanning, "The netflix prize," in *KDD-Cup Workshop*, 2007, pp. 3–6. [Online]. Available: http://su-2010-projekt.googlecode.com/svn-history/r157/trunk/literatura/bennett2007netflix.pdf

[24] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11." *JMLR W&CP*, pp. 3–18, 2012.

[25] R. M. Lawrence Page, Sergey Brin and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1998.

[26] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale Parallel Collaborative Filtering for the Netflix Prize," in *AAIM*, 2008, pp. 337–348.

[27] "Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, circa 2002," http://webscope.sandbox.yahoo.com/.

[28] "Giraph," http://giraph.apache.org/.

[29] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *SIGMOD*, 2014, pp. 827–838.

[30] 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.

[31] 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.

[32] R. Xin, D. Crankshaw, A. Dave, J. Gonzalez, M. Franklin, and I. Stoica, "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics," Tech. Rep., 2014.