TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee,
Min-Soo Kim[1], Jinha Kim, Hwanjo Yu
POSTECH, DGIST[1]
{wshan.postech,syleeDB,realpky85,handolleejh}@gmail.com,
mskim@dgist.ac.kr, {goldbar,hwanjoyu}@postech.ac.kr

ABSTRACT

Graphs are used to model many real objects such as social networks and web graphs. Many real applications in various fields require efficient and effective management of large-scale graph structured data. Although distributed graph engines such as GBase and Pregel handle billion-scale graphs, the user needs to be skilled at managing and tuning a distributed system in a cluster, which is a nontrivial job for the ordinary user. Furthermore, these distributed systems need many machines in a cluster in order to provide reasonable performance. In order to address this problem, a disk-based parallel graph engine called GraphChi, has been recently proposed. Although GraphChi significantly outperforms all representative (disk-based) distributed graph engines, we observe that GraphChi still has serious performance problems for many important types of graph queries due to 1) limited parallelism and 2) separate steps for I/O processing and CPU processing. In this paper, we propose a general, disk-based graph engine called TurboGraph to process billion-scale graphs very efficiently by using modern hardware on a single PC. TurboGraph is the first truly parallel graph engine that exploits 1) *full parallelism* including multi-core parallelism and FlashSSD IO parallelism and 2) *full overlap* of CPU processing and I/O processing as much as possible. Specifically, we propose a novel parallel execution model, called *pin-and-slide*. TurboGraph also provides engine-level operators such as BFS which are implemented under the pin-and-slide model. Extensive experimental results with large real datasets show that TurboGraph consistently and significantly outperforms GraphChi by up to *four orders of magnitude*! Our implementation of TurboGraph is available at “http://wshan.net/turbograph" as executable files.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search Process*; H.2.4 [**Database Management**]: Systems—*Parallel Databases*

General Terms

Algorithm; Performance

Keywords

Graph processing; Big data; Parallelism; Pin-and-slide

1. INTRODUCTION

Graphs are used to model many real objects such as social networks, web graphs, chemical compounds, and biological structures. Many real applications in computer science, bioinformatics [23], chemistry [22], physics, health-care, and geology require efficient and effective management of graph structured data.

The size of real graphs is very large. For example, Facebook reached one billion users on Oct. 4, 2012. Another example is the Yahoo Web graph [1] consisting of 1.4 billion vertices and 6.6 billion edges. Graphs with billions of vertices resident in memory require hundreds of gigabytes of main memory, which is only possible in very expensive servers [14]. For fast graph retrieval on a single commodity PC, graphs must be stored in fast external memory, such as FlashSSDs.

Many scalable systems have been recently proposed to handle big graphs efficiently by exploiting distributed computing. For example, GBase [9, 10] is a recent graph engine using MapReduce. It shows that, if the graph is represented as a compressed matrix, matrix-vector computation solves many representative graph queries including global queries such as PageRank and targeted queries such as induced subgraph and k-step neighbor queries. However, distributed systems based on MapReduce are generally slow unless there is a sufficient number of machines in a cluster. For example, GBase used 100 machines to answer a two-step out-neighbor query from a given vertex in the Yahoo Web graph in about 265 seconds. To solve the inherent performance problem of MapReduce, several distributed systems based on the vertex-centric programming model, such as Pregel [17], GraphLab [16], and PowerGraph [4], have been proposed. In this programming model, the user only needs to write a function for each graph query type without the knowledge of distributed programming, which is invoked for each vertex by the underlying systems. However, efficient graph partitioning in a distributed environment for all types of graph operations is very difficult [14]. Furthermore, the user needs to be skilled at managing and tuning a distributed system in a cluster, which is a nontrivial job for the ordinary user. These distributed systems still need many machines in order to provide good performance.

Recently, a disk-based graph processing engine on a single PC called GraphChi [14] has been proposed to address the problems of the distributed graph systems. GraphChi significantly outperforms all representative (disk-based) distributed graph engines [14]. GraphChi exploits the novel concept of parallel sliding windows (PSW) for handling billion-scale disk-based graphs. GraphChi also uses the vertex-centric programming model. Since GraphChi is a disk-based engine rather than a distributed system, messages passing through edges are implemented as updating values to the edges. PSW divides the vertices into P execution intervals, and each execution interval contains a shard file which stores all edges that have the target vertices in that interval. The edges in each shard file are ordered by their source vertices. PSW processes one shard file at a time. Processing each shard consists of the three *separate* substeps: 1) loading a subgraph from disk; 2) updating the vertices and edges; and 3) writing the updated parts of the subgraph to disk.

We observe that PSW incurs four serious problems. 1) In order to start updating vertices/edges in a shard file, their in-edges must be fully loaded in memory. This property hinders the *overlapping* of the three steps; 2) All edges in the shard file whose source and target vertices are in the same execution interval are processed in *sequential order* [14], which hinders *full parallelism*; 3) At each iteration, a significant number of updated edges can be flushed to disk. If the size of the graph is very large and/or there exist many iterations, GraphChi involves a significant amount of disk I/Os. 4) Even if a query needs to access a small portion of the data graph, it reads the whole graph at the first iteration. In other words, it results in poor scalability in terms of the number of threads and poor utilization in terms of total hardware resources (CPU and flashSSD).

We present a general, disk-based graph engine called TurboGraph to process billion-scale graphs very efficiently by using modern hardware on a single PC. TurboGraph is the first truly parallel graph engine on a single machine that exploits 1) *full parallelism* including FlashSSD IO parallelism and multi-core parallelism and 2) *full overlap* of CPU processing and I/O processing as much as possible. Note that multi-core CPUs can process multiple CPU jobs at the same time, while FlashSSDs can process multiple I/O requests in parallel by using the underlying multiple flash memory packages.

We propose a novel parallel execution model called *pin-and-slide*, which implements the *column view* of the matrix-vector multiplication. By interpreting the matrix-vector multiplication in the column view, we can restrict the computation to just a subset of the vertices, utilizing two types of thread pools, the execution thread pool and the asynchronous I/O callback thread pool (or simply callback thread pool) along with a buffer manager. Specifically, given a set V of vertices, we first identify the corresponding pages for the vertices and then *pin* those pages in the buffer pool. Since we exploit the buffer manager of the storage engine, some pages that were read before can exist in the buffer pool, and we can guarantee that those pages pinned are to be resident in memory until they are explicitly unpinned. We next issue parallel asynchronous I/Os to the FlashSSD for pages which are not in the buffer pool. Note that we do not wait for the completion of those I/O requests. Instead, for those pages already in the buffer pool, multiple execution threads concurrently process vertices in V that are also in the pages pinned and their adjacency lists. At the same time, as soon as the I/O request for each page is completed, a callback thread processes the CPU processing of the page. As soon as either an execution thread or a callback thread finishes the processing of a page, it unpins the page, and an execution thread issues an asynchronous I/O request to the FlashSSD. That is, we *slide* the processing window one page at a time for all pages corresponding to the input vertices. This way, we can fully utilize CPU and FlashSSD I/O parallelism and fully overlap CPU processing and I/O processing.

We also provide parallel, engine-level graph primitives including disk-based matrix-vector computation and breadth-first search, which serve as core graph operations in graph databases. Note that these operators can be efficiently implemented under the *pin-and-slide* model by leveraging the full parallelism and the full overlap.

Our contributions are as follows: 1) We propose a general and scalable graph engine on a single machine which fully exploits multi-core parallelism and I/O parallelism and fully overlaps CPU processing with I/O processing. 2) We propose efficient disk and memory structures for representing billion-scale graphs, which can efficiently support both graph traversal and bitmap-based operations. Note that, when we traverse graphs by accessing adjacency lists, which is a unique requirement in graph databases, the traditional IR approach, which stores "vertex IDs" in the posting list, will eventually fail due to the very large size of the mapping table which maps a vertex ID to the corresponding disk address, although it shows good performance for bitmap-based operations such as bitmap intersection. 3) We propose fast and scalable core graph operations which implement the *pin-and-slide* model. 4) Extensive experimental results with large real datasets show that TurboGraph consistently and significantly outperforms the state-of-the-art methods by up to *four orders of magnitude*! Our implementation of TurboGraph is available at "http://wshan.net/turbograph" as executable files.

The rest of this paper is organized as follows. Section 2 reviews related work. In Section 3, we propose efficient disk and memory structures for storing billion-scale graphs. Section 4 proposes the concept of the *pin-and-slide* model and gives the detailed implementation of the model. Section 5 describes how different types of queries are executed in TurboGraph. We provide empirical evaluations and comparisons using large real graphs in Section 6, and Section 7 summarizes and concludes the paper.

2. RELATED WORK

There is a number of algorithms for a specific type of graph query, *e.g.*, finding neighborhoods [18], community detection [12], finding induced subgraphs [2], computing the number of triangles [8], finding connected components [20], computing subgraph isomorphism [6, 15, 5], and PageRank. Most of them are based on a main memory model that limits their ability to handle large-scale, disk-resident graphs. Thus, they do not tend to scale well for web-scale graphs with billions of vertices and edges. To efficiently handle web-scale graphs and reduce the redundant effort of developing an algorithm for each query, many scalable and high-level graph systems [17, 11, 10, 9, 16, 4, 19, 14] have recently been proposed. They support various kinds of graph queries instead of a specific graph query and also can handle web-scale graphs with billions of vertices and edges. They can be classified into *distributed* systems and *single-machine* systems depending on the size of system, and distributed systems can be further categorized into *synchronous* approaches and *asynchronous* approaches.

Distributed synchronous approaches: PEGASUS [11] and GBase [10, 9] are based on MapReduce and support matrix-vector multiplication using compressed matrices. Pregel [17] is not based on MapReduce but on the vertex-centric programming model where a vertex kernel is executed in parallel on each vertex. In this model, the user only needs to write a function for each graph query type, which is invoked for each vertex by the underlying systems. Pregel follows the Bulk-Synchronous Parallel (BSP) message passing model in which all vertex kernels run simultaneously in a sequence of super-steps. Within a super-step, each kernel receives all messages from the previous super-step and sends them to its neighbors in the next super-step. A *barrier* is imposed between super-steps to ensure that all kernels finish processing messages. All synchronous approaches above could suffer from costly performance penalties since the runtime of each step is determined by the slowest machine in the cluster. Such an imbalance in runtime may be caused by a lot of factors including hardware variability, network imbalances, and power-law degree distributions of natural graphs.

Distributed asynchronous approaches: GraphLab [16] is also based on the vertex-centric programming model but a vertex kernel is executed in asynchronous parallel on each vertex. In GraphLab,

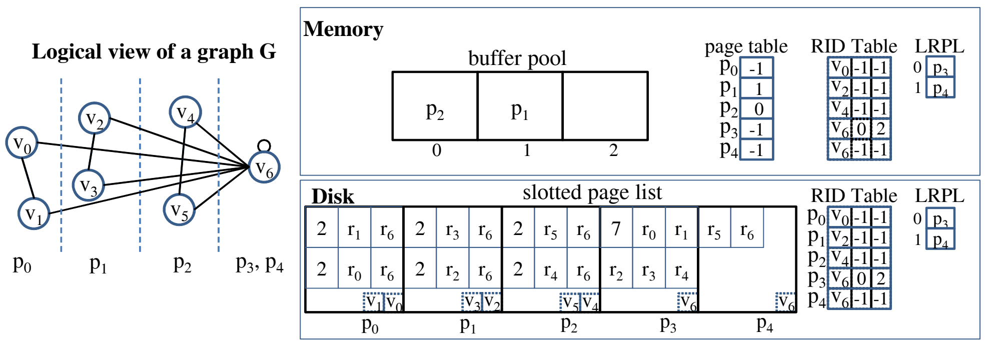


Figure 1: In-memory and disk representation of graphs in TurboGraph.

each vertex reads and writes data on adjacent vertices and edges through shared-memory instead of messages. Since asynchronous systems update parameters using the most recent parameter values as input, they can make many kinds of queries converge faster than synchronous systems do. However, some algorithms based on asynchronous computation require serializability for correctness, and GraphLab allows a user to choose the level of consistency needed for correctness. PowerGraph [4] is basically similar to GraphLab, but it partitions and stores graphs by exploiting the properties of real-world graphs of highly skewed power-law degree distributions. Even though the above asynchronous approaches have the algorithmic benefits of converging faster, efficient graph partitioning in a distributed environment for all types of graph operations is an inherently hard problem [14]. Furthermore, the user must be skilled at managing and tuning a distributed system in a cluster, which is a nontrivial job for the ordinary user. Trinity [19] is a memory-based distributed system, which does not use MapReduce, for faster query processing on graphs. Trinity focuses on optimizing memory and communication cost under the assumption that the whole graph is partitioned across a memory cloud. Although this approach is interesting, it requires an expensive high-bandwidth network hardware in a cluster for good performance.

Single-machine approaches: GraphChi [14] is a disk-based single-machine system following the asynchronous vertex-centric programming model. GraphChi exploits the novel concept called Parallel Sliding Windows (PSW) for handling web-scale disk-based graphs. Since it is a disk-based engine rather than a distributed system, messages passing through edges are implemented as updating values to the edges. PSW divides the vertices into P execution intervals, and each execution interval contains a shard file which stores all edges that have the target vertices in that interval. The edges in each shard file are ordered by their source vertices. PSW processes one shard file at a time. When processing each shard, there are three *separate* sub-steps: 1) loading a subgraph from disk; 2) updating the vertices and edges; and 3) writing the updated parts of the subgraph to disk. Even though GraphChi is very efficient, and thus able to significantly outperform large Hadoop deployments on many graph problems while using only a single machine, there are still four serious problems which we have addressed in Section 1. As a consequence, it results in poor scalability in terms of the number of threads and poor utilization in terms of total hardware resources (CPU and external memory).

3. EFFICIENT GRAPH STORAGE

3.1 Disk-based Graph Representation

A graph database in TurboGraph contains a list of slotted pages where each page size is some multiple of 1 MBytes. Each page consists of consecutively stored *records* (here, a record means an adjacency list) and their *slots*. Each slot, which grows backward from the end of the page, stores a pair consisting of a vertex ID and the start offset of the adjacency list corresponding to that vertex ID. A record ID (RID) consists of a page ID and a slot number.

In the adjacency list, we can store "vertex IDs" by following the convention of the traditional IR approach. This approach is useful for adjacency list intersection. However, we need to convert a vertex ID to its corresponding RID when performing graph traversal since the graph is stored in disk. That is, we must maintain a mapping table for this conversion. However, if there are a billion vertices in the graph database, the size of the mapping table is too large to fit into memory. The other choice is to store RIDs in the adjacency list directly. However, in this case, we need to map a RID to its corresponding vertex ID. For this purpose, we propose the *RID table* whose size is equal to the number of the total pages in the database. Each entry in the RID table stores only the start vertex ID in the page. Since the size of this RID table is very small, we can safely make it resident in memory. The translation of a RID to the corresponding vertex ID can be done in O(1). Given a RID (pageID, slotNo), its vertex ID is calculated as $RIDTable[pageID].startVertexID + slotNo$.

In most cases, the size of the adjacency list is smaller than the size of a single page, so we store multiple adjacency lists in one page, which we call "Small Adjacency" list page (*SA page*). Sometimes, however, the size of the adjacency list of a vertex v may be larger than one page. Then, we must break it into multiple sublists and store them in several pages. Thus, only one adjacency list is stored in those pages. We call such pages "Large Adjacency" list pages (*LA pages*). We assume the LA pages of v are $\{p'_1, \cdots, p'_k\}$. Then, in order to keep the mapping information from v to $\{p'_1, \cdots, p'_k\}$, we maintain a so-called Large Record Page List (*LRPL*). Here, $\{p'_1, \cdots, p'_k\}$ is consecutively stored in the LRPL such that p'_1 is stored at the i-th entry of the LRPL, p'_2 stored at the $i + 1$-th entry, and so on. We also let each entry of the RID table store two additional values — the start offset in the LRPL, and the total number of the LA pages — in addition to the start vertex ID. For example, the entry corresponding to p'_1 in the RID table stores (v, i, k), where v is the start vertex ID, i the start offset in the LRPL, and k the total number of LA pages. The function `GetPages(p)` returns p, if p is an SA page; otherwise, it returns a list of all LA pages for p.

Figure 1 shows an example of the in-memory and disk representation for a data graph G. For the vertices $v_0 \sim v_5$, their adjacency lists are stored as small records in pages $p_0 \sim p_2$, while the adjacency list of v_6 is stored as a large record which spans the two pages p_3 and p_4. The first entry in the RID table is (v_0, -1, -1), which means that the start vertex ID of the first page is v_0, and it is an SA page. The fourth entry corresponding to page p_3 is (v_6, 0, 2), which means that the start vertex of p_3 is v_6, and it is one of the first LA pages, where the start offset in the LRPL is 0, and the number of LA pages is 2. That is, in order to access the pages for the large adjacency list, we first need to access the first and second entries in the LRPL. In this example, `GetPages`(p_0) returns p_0, while `GetPages`(p_3) returns a list $[p_3, p_4]$.

When each vertex has attributes, we vertically partition the attribute values of all vertices as the column-oriented DBMS does [21]. That is, we create a file for each attribute, so that only relevant data can be loaded in memory if necessary. This way, we can increase cache utilization.

The slotted page is known to be very good for supporting efficient updates. That is, when we first build a database, the storage utilization of pages is set to $X\%$ ($\geq$ 50%). Thus, adding edges to adjacency lists in a page can be allowed directly by moving records to the available space within the page and updating the offsets in the corresponding slots accordingly. When the page is full, we move some records to a new page and save their RIDs in the original page. When a moved record needs to be moved to another page, we can store it in a new page and save the new RID in the original page. This way, we can access the moved record with a single indirection at most. If the number of moved records is larger than a given threshold, we reorganize the database. When we delete an edge from an adjacency list, we simply update the corresponding RID in the adjacency list to NULL. If the number of deletes to the page is larger than a given threshold, we compact the page when it needs to be updated.

3.2 In-memory Data Structures and Core Operations

The buffer manager of TurboGraph maintains a buffer pool, actually, an array of *frames*, each of which consists of the page-sized sequences of main memory bytes and some meta information such as pin count, reference bit, and dirty bit. Note that the RID table and the LRPL are resident in memory. The core functions supported by the buffer manager are PINPAGE (pid) and UNPINPAGE (pid). When we invoke PINPAGE for a given page ID, pid, the buffer manager first checks whether the page exists in the buffer. Then, we simply increase its pin-count. Otherwise, it obtains an empty frame by the LRU replacement policy and loads the page from disk to the frame. It then increases the pin count of the page and returns the memory address of the frame where the page was loaded. If pid is one of the first LA pages, TurboGraph pins all its LA pages. UNPINPAGE simply decreases the pin count of the page. If pid is one of the first LA pages, TurboGraph unpins all its LA pages. The traditional buffer manager maintains a hash table to translate the page ID to the memory address of the frame. We observe that this hash table approach could incur significant overhead for the graph operations that follow edges by accessing the RIDs in the adjacency list. Thus, we use a different approach here. Since the page size of TurboGraph is at least one MByte, *i.e.*, large enough, the number of pages required even for billion-scale graphs is small enough that the whole page table can fit in memory. Thus, we use the traditional *page table* approach used in the operating system, which uses an array instead of a hash table. In Figure 1, two pages, p_2 and p_1, are in the buffer pool, and the corresponding entries in the page table have the values of 0 and 1, respectively.

TurboGraph additionally provides a core function, PINCOMPUTEUNPIN(PageID pid, list<RID> $RIDList$, UserObject uo), in order to allow asynchronous I/Os to the FlashSSD. Here, pid is the ID of a page to read from disk, $RIDList$ the list of RIDs to process, and uo, the user object passed by a user. More specifically, uo contains several variables or methods including $Compute$, which is a user-defined function for processing the $RIDList$. Calling this function first reserves, *i.e.*, *pre-pins*, available frames and issues asynchronous I/O requests to the FlashSSD. When the I/O is completed, a callback thread processes the vertices in the $RIDList$ by invoking the user-defined function $uo.Compute$. After processing all vertices in the $RIDList$, the callback thread ***unpins*** the page p_0. For example, in Figure 1, when an execution thread invokes PINCOMPUTEUNPIN(p_0, $[v_1]$, uo), the buffer manager pins one free frame, the third one in the buffer, delivers the asynchronous I/O for p_0 to the FlashSSD, and returns the control to the execution thread so as to continue to execute its remaining task. When the I/O for p_0 is completed, the callback thread invokes $uo.Compute(v_1$, Iterator($v1.adj$)), where the iterator is used to iterate through the adjacency list $v1.adj$ using one of its methods, GETNEXT. After the processing of all requested adjacent lists in p_0 including $v1.adj$, p_0 is unpinned.

4. DISK-BASED PARALLEL GRAPH COMPUTATION

Suppose that a graph $G = (V, E)$ is represented by an adjacency matrix $M(G)$, where v_i is the i-th vertex in G. Let $M(G)_i$ the i-th column vector of $M(G)$. When we have a column vector X ($|X| = |V|$), we can define the matrix-vector multiplication between $M(G)$ and X ($Y = M(G) \times X$) as $Y = \sum_{i=1}^{|V|} M(G)_i \times X_i$ in the *column* view.

Depending on applications, which can define their own multiplication and summation semantics, we can generalize both operators with the user-defined function $Compute$. Without the loss of generality, $M(G)_i$ is represented as $v_i.adj$ which is the adjacency list of V_i.

Unlike the previous approaches in [9, 10] which interpret the matrix-vector multiplication in the row view, by interpreting it in the column view, we can restrict the computation to just a subset of vertices, $v_{I[1]} \sim v_{I[k]}$. The j-th vertex in this subset corresponds to $v_{I[j]}.adj$. Then, this generalized matrix-vector multiplication is represented as in Algorithm 1.

Algorithm 1 Matrix-Vector-Multiplication($G = (V, E), X, I, Y$)

```
1: for $i = 1$ to $|I|$ do
2:     $Compute(v_{I[i]}.adj, X_{I[i]}, Y)$
3: end for
```

The *pin-and-slide* model is a new computing model for efficiently processing this generalized matrix-vector multiplication in the column view. In Section 4.1, we explain the key concept of the *pin-and-slide* model mainly under the assumption that the vectors X and Y are indicator vectors where the values are zero or one. Since X is an indicator vector, we can derive I from X. In Section 4.2, we extend the basic *pin-and-slide* model to support general vectors.

4.1 The Pin-and-Slide Model

A *pin-and-slide* system has a buffer pool, a graph database, and two types of threads: execution threads and callback threads. Whenever a thread issues an asynchronous I/O with a callback function $uo.Compute$ to the buffer manager, the execution control directly returns to the calling thread, and the buffer manager delivers the request to the FlashSSD through OS. On completion of the request, the OS signals any idle callback thread, and the thread invokes $Compute$.

The main task of the *pin-and-slide* model is to efficiently access all relevant adjacency lists and to invoke $uo.Compute$ for them. Before accessing the adjacency list of any vertex v, we must pin the corresponding page, since it may not be resident in the buffer pool. However, calling `PinPage` for each adjacency list incurs nontrivial overhead. Thus, for efficient access to adjacency lists, we should identify the pages containing those adjacency lists first, *i.e.*, the requests for the adjacency lists must be converted into those for their pages. This way, we can invoke just one `PinPage` for each page.

After identifying the pages of interest, we check whether any of them exist in the buffer pool. This step is also important for performance, since the buffer manager may replace those pages according to its replacement policy if we access them arbitrarily. After identifying them, we need to pin them in the buffer pool, so we can guarantee that they are to be resident in memory until they are explicitly unpinned.

A large adjacency list spans across multiple LA pages, and thus, we need to pay attention to them when we pin the large adjacency list. For instance, after pinning some of the LA pages, if the current thread is unable to find any available frame unpinned in the buffer pool, we must wait until the other threads unpin some pages. To avoid this problem, we first identify partially loaded LA pages (*PL pages*) before pinning them. Only after all LA pages for a large adjacency list are fully loaded, we safely pin them. Here, if there are multiple, partially loaded, large adjacency lists, we do not invoke PINCOMPUTEUNPIN for them arbitrarily. Instead, we select the adjacency lists which could maximize the performance. This poses an interesting question: "Which LA pages should be loaded and pinned first to maximize the buffer utilization?"

We model this as an application of the 0-1 knapsack problem. An *item* corresponds to the PL pages for a large adjacency list, and the *benefit* for the item is the number of the total LA pages for the adjacency list. The *size* of the item is defined as (the number of the total LA pages $-$ the number of its PL pages). The size of the knapsack corresponds to the number of unpinned pages in the buffer pool. The goal is to maximize the benefit of all items in the knapsack. It is well-known that the greedy solution adding the items into the knapsack in decreasing order of the benefit-to-size ratio until the knapsack is full, is a good approximated solution to this problem. For those PL pages in the knapsack, we issue parallel I/Os to the FlashSSD for the remaining pages to be loaded.

After this step, if there are available unpinned pages in the buffer pool, the main thread issues asynchronous I/O requests to the FlashSSD. Next, for those pages pinned, multiple execution threads including the main thread concurrently process vertices and their adjacency lists in the buffer pool by executing the user-defined function. At the same time, as soon as the I/O request for each page is completed, a callback thread processes the vertices and their adjacency lists in the page. As soon as either an execution thread or a callback thread finishes the processing of a page, it unpins it, and an execution thread issues an asynchronous I/O request to the FlashSSD. Here, if the page is the first LA page, we unpin all LA pages, since there is only one vertex for those pages, and the vertex has been processed completely. In this way, we slide the processing window one page (or multiple pages in the case of LA pages) at a time for all pages corresponding to the input vertices. In this way, we can fully utilize CPU and FlashSSD I/O parallelism and fully overlap CPU processing and I/O processing.

Algorithm 2 shows the detailed steps of our *pin-and-slide* model. Its inputs are a bit vector for X and a user object uo which contains a user-defined function $Compute$ as one of its methods. The output is a bit vector for Y, which will be generalized in Section 4.2. Since we use bit vectors for the graph, we can support billions of vertices with just a small amount of memory. For example, to support 1.4 billion vertices of the Yahoo Web graph, the size of the bit vector is only 168.5 MBytes. The algorithm initializes three concurrent vectors $SAPList$, $LAPList$, and $pinnedList$. The algorithm first groups X by page ID and obtains a list of page IDs, $PList$ (Line 2). Next, we identify a list of page IDs of partially loaded, large adjacency lists, $PLPList$ (Line 3). In Lines 5 $\sim$ 11, we pin already loaded pages, which are SA or LA pages, in the buffer pool. In Lines 12 $\sim$ 18, we issue parallel asynchronous I/Os for a $PLPList$ according to the 0-1 knapsack algorithm we have explained. Here, `GetRIDList`(pid, X) returns a list of RIDs whose vertex IDs are masked in X. As soon as those pages are loaded, the callback threads process them. Next, we find two lists of page IDs, $SAPList$ and $LAPList$, one for small adjacency lists and the other for the large ones (Line 19). Then, we sort the LAPList by the number of LA pages in ascending order, so that we read a smaller set of LA pages first in the next step (Line 20). If there are available frames, we issue parallel asynchronous I/Os by calling ISSUEPARALLELREAD, which is described in Function 3 (Line 22). We note that ISSUEPARALLELREAD uses a latch free approach which exploits hardware-level atomic operations for consistency and performance. For those pages pinned in Lines 5 $\sim$ 11, the multiple execution threads access the adjacency lists in the pages pinned (Lines 23 $\sim$ 29). After processing each page, we un-

Algorithm 2 PINANDSLIDE (BitVector X, UO uo, BitVector Y)

```
1: Variable ConcurrentVector $SAPList, LAPList, pinnedList$
2: group X by page ID into PList; /* when an adjacency list is large, its
   first LA page is in the PList*/;
3: identify a list PLPList of partially loaded LA pages from PList;
4: PList = PList - PLPList;
5: for each PageID $pid$ in PList do
6:     if (GetPages($pid$) are in the buffer) /* if they are fully loaded*/ then
7:         PinPage(pid); /* only pinning */
8:         PList.remove(pid);
9:         pinnedList.insert(pid);
10:    end if
11: end for
12: order PLPList by $\frac{benefit}{size}$ /*Here, we use the knapsack principle.*/
13: for each PageID $pid$ in PLPList do
14:     if (# of pages to load more for page $pid$ <= # of unpinned pages in
        the buffer) then
15:         PinComputeUnpin($pid$, GetRIDList(GetPages($pid$), X), uo);
            /*note that Y can also be accessed by uo.Y*/
16:         PLPList.remove(pid);
17:     end if
18: end for
19: divide PList●PLPList into SAPList and LAPList; /*● is the list con-
    catenation operator*/
20: sort LAPList by |GetPages($pid$)| $s.t.$ $pid \in$ LAPList in ascending or-
    der;
21: PList ← SAPList●LAPList;
22: IssueParallelRead(PList, X, uo, Y);
23: parallel for each PageID pid in pinnedList do
24:     parallel for each vertex v in GetRIDList(pid, X) do
25:         uo.Compute(v, iterator(v.adj), Y);
26:     end parallel for
27:     UnpinPage(pid);
28:     IssueParallelRead(PList, X, uo, Y); /*sliding*/
29: end parallel for
30: while |PList| > 0 do
31:     IssueParallelRead(PList, X, uo, Y); /*pin-and-slide for the remain-
        ing pages*/
32: end while
```

Algorithm 3 ISSUEPARALLELREAD(PageList $PList$, BitVector X, UO uo, BitVector Y)

```
atomic {
    while (# of unpinned pages in the buffer pool > 0) do
        remove PageID pid from PList;
        PinComputeUnpin($pid$, GetRIDList(GetPages($pid$), X), uo); /* is-
        sue parallel I/O*/
    end while
}
```

pin it and issue parallel asynchronous I/Os by calling ISSUEPARALLELREAD (Line 28). After consuming all pre-pinned pages, the main thread repeatedly invokes ISSUEPARALLELREAD for the remaining pages, while the callback threads process those requested pages in parallel (Lines 30 $\sim$ 32).

4.2 Handling General Vectors

In this section, we explain how we handle general vectors (as opposed to indicator vectors) in the *pin-and-slide* model. For example, the input vector for PageRank is an indicator vector having all 1s, which means we compute PageRank values for all vertices, but the output vector for PageRank is the vector storing the PageRank values of all vertices at the current iteration. The size of the general vector may be too large to fit in memory. We call this kind of general vector *large vector*.

In addition, for any given vertex v, the user-defined function $uo.Compute$ may need to access the attribute values of adjacent vertices of v. For example, for the incoming adjacent vertices of v, the $uo.Compute$ function for PageRank needs to access two large vectors, one for the out-degree values and the other for the PageRank values at the previous iteration. We note that the access pattern for these large vectors is random rather than sequential. We call these kinds of vectors *random vectors*. Conversely, since the access pattern for the input vector is sequential, we call this kind of vector *sequential vector*. Note that, depending on the application, the output vector can be a random vector as well. As mentioned in Section 3, a file exists for each vertex attribute. In our graph database, one large vector is created in disk for each attribute of the vertex.

Now, we generalize the PINANDSLIDE algorithm so as to handle large general vectors. The main idea is to adopt the concept of the *block-based nested loop join*, which is well-known in the database area. We regard the set of pages pinned in the current buffer as a *block* and also assume that a general vector is partitioned into multiple *chunks* such that each chunk fits in memory. Then, we "join" a block in the buffer with a chunk of each random attribute vector, and save the results in the corresponding chunk of the output vector. (We explain the exact meaning of "join" later.) Note that the memory area for the large vector is separately allocated in TurboGraph, to minimize the buffering effect for these vectors.

After issuing ISSUEPARALLELREAD at Line 22, we know the range for the output vector that corresponds to the range of graph vertices pinned in the buffer pool. Here, we allocate a memory space, which is typically much smaller than the size of those pages pinned. For example, the size of each element in the output vector storing a PageRank value is four bytes for a float type or eight bytes for a double type, which is much smaller than the average size of the adjacency lists.

In order to join the current block with the first chunk of each random vector, we need to read the first chunk of each random vector before invoking PINCOMPUTEUNPIN. Therefore, in Line 15, we only secure free frames for partially loaded LA pages and read the first chunk of each random vector before Line 22. After that, we issue asynchronous I/Os for those partially loaded pages. Note that, in Lines 30$\sim$32, the callback threads join the remaining blocks with the random vector. Thus, we also need to read the first chunk of each random vector before Line 31.

Now, we explain how to "join" the current block with the remaining chunks of each vector. When we join the current block with the first chunk of each vector, at Line 25, some adjacent vertices of v may not be in the range of the first chunk of each random vector. Thus, the user-defined function may not properly complete its job for those adjacent vertices. Therefore, we need a systematic approach to solve this problem. In TurboGraph, we pass the range R of the current chunk to the user-defined function, and the function only accesses the attribute values of the adjacent vertex whose ID is in the range R. Here, the user-defined function invokes GETNEXT to iterate through the adjacency list. GETNEXT returns a special return code `ON_GOING` when the current adjacent vertex is out of the range R. If the return code of GETNEXT is `ON_GOING`, the user-defined function stops processing the adjacency list and returns to the calling thread with the special return code. The main thread pushes into a queue those iterators having the special return codes. Now, the queue is regarded as a block, and thus, we can join this block with the remaining chunks. We repeat this process until the queue is empty. This additional task needs to be executed before Line 30 and after Line 31.

Now, we explain when we can slide the execution window of the *pin-and-slide* model for handling large vectors. When we finish processing the last chunk of the random vector, we can safely slide the execution window by the size of the pinned pages in the buffer.

Example 1. *We explain our* pin-and-slide *model handling general vectors by using a PageRank query. Figure 2 illustrates all steps in the* pin-and-slide *model for the first iteration of PageRank. We assume the same graph database as Figure 1. There are two large attribute vectors, outDegree and prevPR, where the former stores the out-degree values of vertices, and the latter the PageRank values at the previous iteration. At the very beginning, all elements of prevPR are just* $\frac{1}{|V|} = \frac{1}{7} \approx 0.143$. *The elements of the large output vector are initialized with zero.*

- *Step 1: After first reading pages from disk into the buffer (e.g.,* $\{p_0, p_1, p_2\}$*), we read the first chunk of each attribute vector into memory. Then, we join between* $block1$ *and* $chunk1$*, i.e., apply* $uo.Compute(v_i, Iterator(v_i.adj))$ *to all pairs of* v_i *and available* $v_j \in v_i.adj$*, which exist in* $block1$ *and* $chunk1$*. Then, we write the computation results, which are not complete yet, to the corresponding* $chunk1$ *of the output vector.*
- *Step 2: We read* $chunk2$ *of each attribute vector into memory, join between* $block1$ *and* $chunk2$*, and update the results of* $chunk1$ *of the output vector, which now has complete values.*
- *Step 3: Since we complete the processing for* $block1$*, we read new pages from disk (e.g.,* $\{p_3, p_4\}$*), i.e., do sliding to* $block2$*. Then, we join between* $block2$ *and* $chunk1$ *and write the results to* $chunk2$ *of the output vector.*
- *Step 4: We do the final join and update* $chunk2$ *of the output vector.*

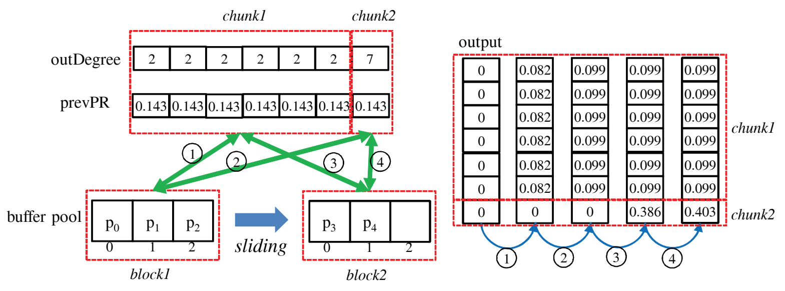
After the above iteration, we obtain the PageRank values of [0.099 , 0.099 , 0.099 , 0.099 , 0.099 , 0.099 , 0.403]*, which are used in the next iteration.*

5. PROCESSING GRAPH QUERIES

In this section, we describe query execution in TurboGraph, which, like GBase, supports both *targeted* and *global* queries. The targeted queries traverse only on a small fraction of the graph, while the global queries traverse the whole graph [9]. As targeted queries, GBase has formulated *seven* different types of queries, which include neighborhood, induced subgraph, egonet, K-core, and cross-edges. We describe those seven targeted queries by using our *pin-and-slide* model, whereas GBase describes the queries by using their matrix-vector multiplication model.

5.1 Targeted Queries

Our PINANDSLIDE algorithm can be regarded as a *full parallelism* and *full overlap* version of a single step of the breadth-first-search (BFS) operation. Therefore, we denote the PINANDSLIDE



$$uo.Compute(v_i, Iterator(v_i.adj)) = 0.85\Sigma_{j\in v_i.adj}\frac{\text{prevPR[j]}}{\text{outDegree[j]}} + \frac{0.15}{|V|}$$

Figure 2: An example of a PageRank iteration in the PinAndSlide algorithm.

algorithm as $BFS_f(V_q)$, where V_q is an input set of vertices, and f is a user-defined function. For a given $G = (V, E)$, BFS_f applies the user-defined function f to every edge $e = (v, w)$ *s.t.* $v \in V_q \wedge (v, w) \in E$ and then returns a union of outputs of f, *i.e.*, $BFS_f(V_q) = \bigcup_{v\in V_q \wedge (v,w)\in E} f((v, w))$. There could be various kinds of user-defined functions. In this paper, we introduce three user-defined functions for describing seven targeted queries in Equations (1), (2) and (3). The term f_{nh} is usually used for describing neighborhood queries, f_{wnh} is used for describing egonet queries, and f_{is} is used for describing induced subgraph queries.

$$f_{nh}((v, w)) = \{w\} \quad (1)$$

$$f_{wnh}((v, w)) = \{v, w\} \quad (2)$$

$$f_{is}((v, w)) = \begin{cases} \{(v, w)\}, & v \in V_q \wedge w \in V_q; \\ \{\}, & \text{otherwise.} \end{cases} \quad (3)$$

In terms of implementation, the input set of vertices V_q is passed as a bit vector, and the output of the BFS_f operator is also set on the same bit vector.

T1: 1-step neighbors. The first query is to find 1-step (out-)neighbors of a query vertex v. The 1-step (in-)neighbors of v can be obtained in the same way by using the adjacency list of in-edges instead of out-edges. We can find this by using the BFS_f operation as follows:

$$NH^1(v) = BFS^1_{f_{nh}}(v). \quad (4)$$

T2: K-step neighbors. The next query is to find k-step (out-)neighbors of a query vertex v. The k-step neighbors of v, $NH^k(v)$ can be defined recursively by $(k-1)$-step neighbors of v, $NH^{k-1}(v)$ as follows:

$$NH^k(v) = BFS^k_{f_{nh}}(v) = BFS^1_{f_{nh}}(BFS^{k-1}_{f_{nh}}(v)). \quad (5)$$

T3: Induced subgraph. The next query is to find an induced subgraph for a given set of vertices $V_q \subseteq V$, which is defined as a graph whose vertices are all in V_q and whose edges are also adjacent in G. We can find it simply by using the user-defined function f_{is} as follows :

$$IS(V_q) = BFS^1_{f_{is}}(V_q). \quad (6)$$

T4: 1-step egonet. The 1-step egonet of a vertex v is defined as the induced subgraph that includes v and its 1-step neighbors. Thus, we can find it easily by using both f_{nh} and f_{is} as follows:

$$EG^1(v) = BFS^1_{f_{is}}(BFS^1_{f_{wnh}}(v)) \quad (7)$$

where the user-defined function $f_{wnh}(v)$ is used for including both v and its 1-step neighbors, *i.e.*, 'within 1-step neighbors.'

T5: K-step egonet. The k-step egonet of a vertex v is defined as the induced subgraph that includes from v to its k-step neighbors, *i.e.*, 'within k-step neighbors.'

$$EG^k(v) = BFS^1_{f_{is}}(BFS^k_{f_{wnh}}(v)). \quad (8)$$

T6: K-core. K-core of a graph means a maximal connected subgraph where all vertices have a degree of at least K. This requires a preprocessing step and a global query with an induced subgraph query as follows:

1. Find a set C of vertices with a degree $\geq K$ from G.
2. Compute $G_K = IS(C)$.
3. Find connected components of G_K, where each is a K-core.

T7: Cross-edges. The cross edges mean the edges crossing between two disjoint sets of vertices V_1 and V_2. They can be computed in a similar way as GBase as follows :

1. Compute three induced subgraphs: $IS(V_1)$, $IS(V_2)$, $IS(V_1 \cup V_2)$.
2. Let E_1, E_2, and E_{12} be the set of edges in $IS(V_1)$, $IS(V_2)$, and $IS(V_1 \cup V_2)$, respectively. The cross edges are $E_{12} - E_1 - E_2$.

5.2 Global Queries

Under our *pin-and-slide* model, global queries are performed by repeated execution of the PinAndSlide algorithm with an initial input vector having all 1s. The *pin-and-slide* model supports a wide range of global queries. It covers all global queries that GBase [10, 9] supports, *i.e.*, degree distribution, PageRank, Random Walk with Restart(RWR), radius estimations, and discovery of connected components. We have already explained briefly how our model processes the PageRank query in Example 1. In addition, it also supports global queries corresponding to matrix-matrix multiplication such as counting triangles, which is beyond the scope of this paper. Our contribution here is that our model significantly improves the performance of global queries, which requires accessing entire billion-scale graphs, as well as targeted queries, even though it runs in just a single machine.

6. EXPERIMENTS

We evaluate the performance of TurboGraph compared with the state-of-the-art graph processing engine, GraphChi. We choose GraphChi since it significantly outperforms existing (disk-based) distributed graph engines [14]. We use three types of queries, breadth-first search, targeted queries, and global queries. As targeted queries, we use 1-step out-neighbor query, 2-step out-neighbor query, and egonet query as in [9, 10]. We measure the average elapsed time of five randomly selected query nodes as in [9, 10]. As global queries, we use the PageRank query and connected component query, where we report the top-20 PageRank values after computing all PageRank values and the number of connected components, respectively.

6.1 Experimental Setup

We use three real datasets for the experiments, LiveJournal [3], Twitter [13], and YahooWeb[1]. The LiveJournal dataset is about an online social networking site with 4.8M vertices and 69M directed edges. The Twitter dataset contains 42M vertices and 1.5B edges. The YahooWeb dataset contains a web graph from Yahoo! with 1.4B vertices and 6.6B edges.

We conduct all the experiments on the same PC with Intel i7 6-core 3.2GHz CPU and 12 Gbytes DRAM. This PC is equipped with two 512GB SSDs of Samsung 840 Series, one for running Windows 7 and the other for running Ubuntu Linux. Note that

TurboGraph can be compiled in Windows, while GraphChi can be compiled in Linux. Considering that disk I/O performance in Ubuntu is better than that in Windows 7[1], we expect that the performance gap between TurboGraph and GraphChi would increase in the same Linux environment. We plan to port our source code to the Linux environment.

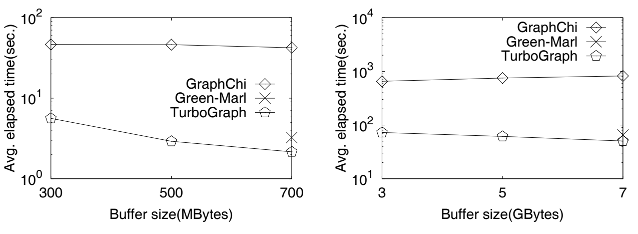
We use six execution threads as a default setting for all compared systems. We disable hyper-threading, so that the maximum number of hardware threads is six.

6.2 Breadth-First Search

The purpose of this experiment is to show whether our *pin-and-slide* model efficiently supports breadth-first search by varying the buffer size and the number of execution threads. If there is sufficient memory available, one may use a fast in-memory based graph engine. Thus, we additionally perform experiments with a state-of-the-art in-memory graph BFS engine Green-Marl [7] which optimizes the BFS query. Note that the in-memory graph engines can be typically implemented much faster than disk-based ones, since there is no translation between memory address and disk address.

6.2.1 Varying the Buffer Size

Figure 3 shows the elapsed times for LiveJournal and Twitter by varying the buffer size. We could run Green-Marl with the buffer sizes of 700 MBytes and 7 GBytes for LiveJournal and Twitter, respectively. Note that TurboGraph can execute BFS queries for YahooWeb in a reasonable time, while Green-Marl failed due to lack of memory, and GraphChi did not finish in a reasonable time. Thus, we report the results for LiveJournal and Twitter. Note that the Y-axis is in log scale. We use these buffer sizes for the remaining experiments. TurboGraph outperforms GraphChi by up to 19.62 and 16.31 times for LiveJournal and Twitter, respectively.



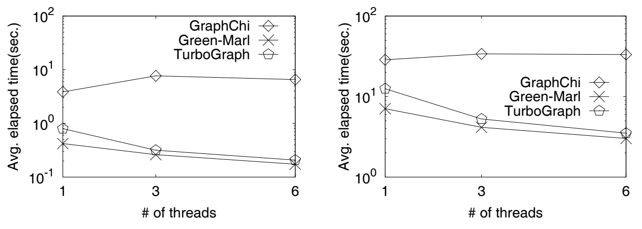
(a) LiveJournal. (b) Twitter.

Figure 3: Varying buffer size.

Note that the database sizes of TurboGraph for LiveJournal and Twitter are about 710M bytes and 12 GBytes, respectively. As the buffer size increases, the *pin-and-slide* model of TurboGraph utilizes the buffer pool in a very smart way and improves the performance accordingly as the buffer size increases. However, lacking this kind of feature, GraphChi shows results indicating that its performance remains almost constant regardless of the buffer size. Green-Marl first loads the whole graph in memory and executes BFS. Although Green-Marl is specially designed for in-memory graph engines, TurboGraph outperforms Green-Marl by a small margin. Note that Green-Marl can not support billion-scale graphs due to its large memory requirement.

6.2.2 Varying the Number of Execution Threads

Figure 4 shows the elapsed times for LiveJournal and Twitter by varying the number of execution threads. For this purpose, we need to pre-load the whole graph in memory for TurboGraph and Green-Marl. However, it is very hard to pre-load the graph for GraphChi. GraphChi requires making only one shard file to pre-load the whole graph into memory. However, if there is only one shard file, GraphChi processes all edges serially. Therefore, instead of making one shard file, we use the standard sharding mechanism, run experiments, and exclude the elapsed time for the disk I/Os from the total elapsed time. The authors of GraphChi took the same approach [14] to analyze the performance for varying the number of execution threads.



(a) LiveJournal. (b) Twitter.

Figure 4: Varying # of execution threads.

Both TurboGraph and Green-Marl increase the BFS performance as the number of execution threads increases. As we see in the figure, TurboGraph achieves better speedups than Green-Marl, although Green-Marl marginally outperforms TurboGraph since it is an in-memory graph engine. When we test with an eight-core PC, TurboGraph marginally outperforms Green-Marl. On the other hand, GraphChi shows poor performance as we increase the number of execution threads. The similar phenomenon has also been reported in [14]. This results show that, TurboGraph fully utilizes multi-core parallelism, while GraphChi does not.

6.3 Targeted Queries

Figure 5 shows the elapsed times for the three targeted queries. TurboGraph significantly outperforms GraphChi for the targeted queries by up to four orders of magnitude. For the YahooWeb dataset, the elapsed times of GBase for one-step out-neighbor query, two-step out-neighbor query, and the egonet query were about 100, 265, and 395 seconds, respectively, when it used 100 machines. Note that the elapsed times of TurboGraph for these queries are about 0.18, 0.47, and 0.58 seconds. Since these targeted queries access a small fraction of the graph, TurboGraph can directly traverse those targeted vertices. Thereby, TurboGraph outperforms GraphChi by up to 10830.48, 1940.90, and 1573.62 times for one out-neighbor query, two out-neighbor query, and the egonet query, respectively.

6.4 Global Queries

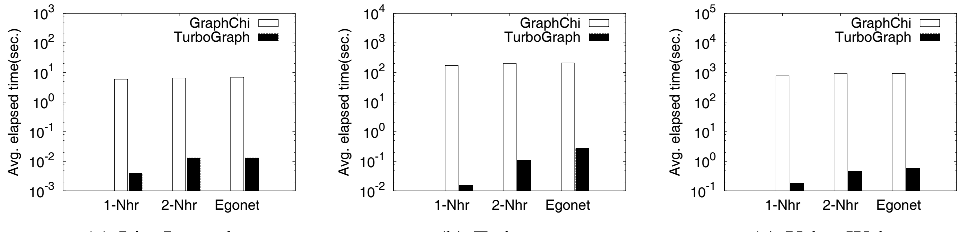
Figure 6 shows the elapsed times for the two global queries, PageRank (10 iterations for LiveJournal and Twitter and one iteration for YahooWeb) and connected component queries. Note that these queries in GraphChi are implemented by the authors of GraphChi. TurboGraph outperforms GraphChi for the PageRank query by up to 27.69 times. This shows that the *pin-and-slide* model of TurboGraph efficiently processes the general large vectors. Note that the Y-axis in Figure6(c) is in log scale.

For counting the number of connected components in the YahooWeb dataset, GraphChi requires more than 265 rounds of iterations, while TurboGraph can count the number of connected components by scanning the whole graph just once, by exploiting the *pin-and-slide* model using a large vector that stores component IDs. Therefore, TurboGraph outperforms GraphChi by 144.11 times.

7. CONCLUSION

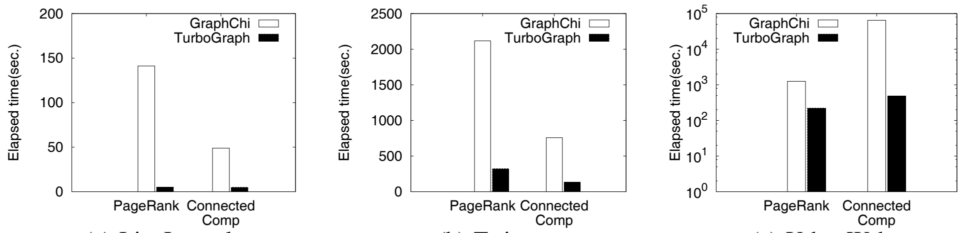
In this paper, we presented a fast, parallel graph engine called TurboGraph for efficiently processing billion-scale graphs on a single PC. We proposed a notion of the *pin-and-slide* model which implements the column view of the matrix-vector multiplication. It utilizes two types of threads, execution threads and callback threads,

[1] http://www.phoronix.com/scan.php?page=article&item=ubuntu_win7_ws&num=4



(a) LiveJournal. (b) Twitter. (c) YahooWeb.

Figure 5: Average elapsed time (Targeted queries).



(a) LiveJournal. (b) Twitter. (c) YahooWeb.

Figure 6: Average elapsed time (Global queries).

along with a buffer manager. In this model, we first identify the corresponding pages for the query vertices and then pin those pages in the buffer pool. We next issue parallel asynchronous I/Os to the FlashSSD for pages which are not in the buffer pool. Without waiting for the completion of those I/O requests, multiple execution threads concurrently process vertices that are in the pages pinned and their adjacency lists. As soon as the I/O request for each page is completed, a callback thread performs the CPU processing of the page. As soon as either an execution thread or a callback thread finishes the processing of a page, it unpins the page, and an execution thread issues an asynchronous I/O request to the FlashSSD.

In addition, we provided parallel, engine-level graph primitives including disk-based matrix-vector computation, which serve as core graph operations in graph databases. These operators have been efficiently implemented under the *pin-and-slide* model by leveraging the full parallelism and the full overlap.

Through extensive experiments on large, real graphs, including billion-node graphs, we showed that TurboGraph outperforms the state-of-the-art algorithms by up to four orders of magnitude. Overall, we believe we provide comprehensive insight and a substantial framework for future research.

8. ACKNOWLEDGEMENTS

This work was supported by Mid-career Researcher Program through NRF grant funded by the MEST (NRF-2011-0016520, NRF-2011-0016029). This work was done while the first four authors were at KNU.

9. REFERENCES

[1] Yahoo webscope. yahoo! altavista web page hyperlink connectivity graph. http://webscope.sandbox.yahoo.com.

[2] L. Addario-Berry, W. S. Kennedy, A. D. King, Z. Li, and B. A. Reed. Finding a maximum-weight induced k-partite subgraph of an i-triangulated graph. *Discrete Applied Mathematics*, 158(7):765–770, 2010.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54. ACM, 2006.

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

[5] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{ISO}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, 2013.

[6] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. igraph: a framework for comparisons of disk-based graph indexing techniques. *Proc. VLDB Endow.*, 3(1-2):449–459, Sept. 2010.

[7] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ASPLOS*, pages 349–362, 2012.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[9] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD*, pages 1091–1099, 2011.

[10] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. gbase: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

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

[12] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[13] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[15] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

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

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[18] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD*, pages 533–542, 2010.

[19] B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical Report 161291, Microsoft Research, 2012.

[20] Y. Shiloach and U. Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[21] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[22] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[23] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.