Fast Iterative Graph Computation: A Path Centric Approach

Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin
Services Computing Technology and System Lab.
Cluster and Grid Computing Lab.
School of Computer Science & Technology
Huazhong University of Science and Technology
Wuhan, China
Email: {ppyuan, hjin}@hust.edu.cn

Ling Liu, Kisung Lee
Distributed Data Intensive Systems Lab.
School of Computer Science
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
Email: lingliu@cc.gatech.edu

***Abstract*—Large scale graph processing represents an interesting systems challenge due to the lack of locality. This paper presents PathGraph, a system for improving iterative graph computation on graphs with billions of edges. Our system design has three unique features: First, we model a large graph using a collection of tree-based partitions and use path-centric computation rather than vertex-centric or edge-centric computation. Our path-centric graph parallel computation model significantly improves the memory and disk locality for iterative computation algorithms on large graphs. Second, we design a compact storage that is optimized for iterative graph parallel computation. Concretely, we use delta-compression, partition a large graph into tree-based partitions and store trees in a DFS order. By clustering highly correlated paths together, we further maximize sequential access and minimize random access on storage media. Third but not the least, we implement the path-centric computation model by using a scatter/gather programming model, which parallels the iterative computation at partition tree level and performs sequential local updates for vertices in each tree partition to improve the convergence speed. We compare PathGraph to most recent alternative graph processing systems such as GraphChi and X-Stream, and show that the path-centric approach outperforms vertex-centric and edge-centric systems on a number of graph algorithms for both in-memory and out-of-core graphs.**

Keywords*—*Graph; Iterative computation; Path; Computing model; Storage

I. Introduction

The number of highly connected datasets has exploded during the last years. Large scale graph analysis applications typically involve datasets of massive scale. Most of existing approaches address the iterative graph computation problem by programming and executing graph computation using either vertex centric or edge centric approaches. The main problems with the existing approaches are two folds.

First, most of the iterative graph computation algorithms need to traverse the graph from each vertex or edge to learn about the influence of this vertex/edge may have over other vertices/edges in the graph. Vertices or edge traversal produces an access pattern that is random and unpredictable. Executing algorithms that follow vertices or edges inevitably results in random access to the storage medium for the graph and this can often be the determinant of performance, regardless of the algorithmic complexity or runtime efficiency of the actual algorithm in use. It induces many cache misses and requires much time to reach the convergence condition. However, fast convergence of the iterative computation on the massive datasets is essential for these applications.

Second, when a large scale graph does not fit into the main memory, we need to partition the graph into multiple chunks such that we can load chunks from disks or SSDs to the main memory in an iterative fashion to finish the whole graph computation. With vertex or edge centric computation models, random hash based partitioning of a graph by vertices or edges becomes the most commonly used strategy for large-scale graph processing systems, because random hash partitioning can produce balanced partitions in terms of vertices or edges. However, this approach breaks connected components of graphs into many small disconnected parts, and thus has extremely poor locality [7], which causes large amount of communication across partitions.

To address these two problems, we develop a path-centric graph computation model, and a path-centric graph processing system − PathGraph for fast iterative computation on large graphs with billions of edges. Our development is motivated by our observation that most of the iterative graph computation algorithms share three common processing requirements: (1) For each vertex to be processed, we need to examine all its outgoing or incoming edges and all of its neighbor vertices. (2) The iterative computation for each vertex will converge only after completing the traversal of the graph through the direct and indirect edges connecting to this vertex. (3) The iterative computation of a graph converges when all vertices have completed their iterative computation.

The PathGraph system design has three unique features: First, we model a large graph using a collection of tree-based partitions. We develop a path-centric graph computation model and show that it is much more efficient than vertex-centric or edge-centric computation. Our path-centric graph parallel computation model significantly improves the memory and disk locality for iterative computation algorithms on large graphs. Second, we design a compact storage that is optimized for iterative graph parallel computation. Concretely, we use delta-compression, partition a large graph into tree-based partitions and store trees in a DFS order. By clustering highly correlated paths together, we further maximize sequential access and minimize random access on storage media. Third but not the least, we implement the path-centric computation model by using a scatter/gather programming model, which parallels the iterative computation at tree partition level and performs sequential local updates for vertices in each tree partition to improve the convergence speed. We compare PathGraph to most recent alternative graph processing systems such as GraphChi and X-Stream, and show that the path-centric approach outperforms vertex-centric and edge-centric systems on a number of graph algorithms for both in-memory and out-of-core graphs.

The rest of the paper is organized as follows. Section II gives the basic concepts and highlight of our PathGraph design. We present locality based graph partitioning scheme and path-centric parallel computing model in Section III and the path-centric compact storage structure in Section IV. In Section V, we introduce how our path-centric computation model and the path-centric storage structure can effectively work together to provide fast convergence. We evaluate PathGraph on real graphs with billions of edges using a set of graph algorithms in Section VI. We briefly review the related work in Section VII and conclude the paper in Section VIII.

II. Overview of PathGraph

PathGraph is a path-centric graph processing system. It models a large graph using a collection of tree-based partitions. PathGraph implements the path-centric computation model by using a scatter/gather programming model, which parallels the iterative computation at partition tree level and performs sequential local updates for vertices in each tree partition to improve the convergence speed.

In this section we first present the basic concepts used in PathGraph and then give an overview of the PathGraph approach. We will present the technical description of the PathGraph development in the subsequent sections.

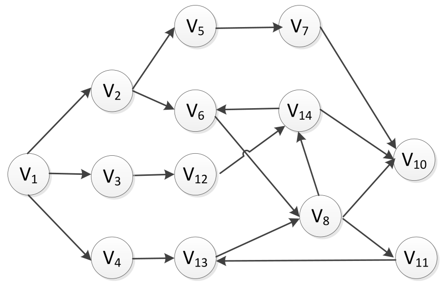
A. Basic Reference Model

PathGraph can handle both directed graphs and undirected graphs. Since undirected graphs can be converted into directed connected graphs and disconnected graphs can be split into connected subgraphs, thus, we will mainly discuss directed connected graph in the following.

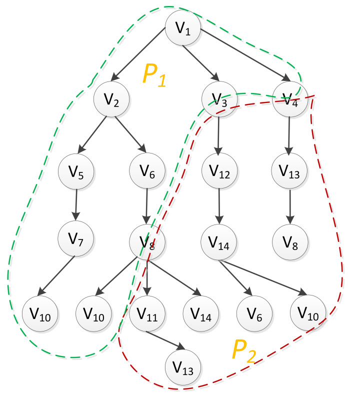
Definition 2.1 (Graph): A graph $G = (V, E)$ is defined by a finite set V of vertices, $\{v_i | v_i \in V\}$ $(0 \le i < |V|)$, and a set E of directed edges which connect certain pairs of vertices, and $E = \{e = (u, v) \in E | u, v \in V\}$. We call u the source vertex of e and v the destination vertex of e.

Definition 2.2 (Path): Let $G = (V, E)$ denote a graph. A path between two vertices u and w in V, denoted by $Path(u, w) =< v_0, v_1, ..., v_k >$, is defined by a sequence of connected edges via an ordered list of vertices, $v_0, v_1, ..., v_k$ $(0 < k < |V|)$, such that $v_0 = u, v_k = w, \forall i \in [0, k-1] : (v_i, v_{i+1}) \in E$. When a path from u to w exists, we say that vertex w is reachable by u through graph traversal.

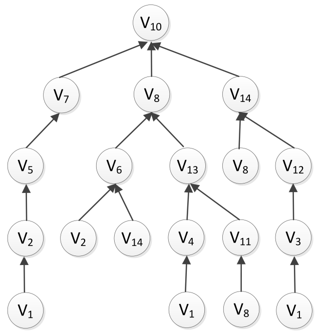
Given that a fair amount of iterative graph computation algorithms need to examine each vertex by traversal of the graph. For example, each iteration of the PageRank algorithm will update a vertex rank score by gathering and integrating the rank scores of the source vertices of its in-edges or scattering



(a) DAG



(b) Forward Edge Traversal Tree



(c) Reverse Edge Traversal Tree

Fig. 1. A DAG and Its Forward-edge Traversal Tree, Reverse-edge Traversal Tree

its rank score to all destination vertices of its out-edges. We below introduce the concept of forward-edge traversal tree and reverse-edge traversal tree, which are two of the core concepts in the path centric computation model of PathGraph.

Definition 2.3 (Forward-Edge Traversal Tree): A forward-edge traversal tree of G, denoted by $T = (V_t, E_t)$, is defined as follows: 1) $V_t \subseteq V$ and $E_t = E$; (2) There is one vertex, say $v_{rt} \in V_t$, $deg^{-}(v_{rt}) = 0$. We call this vertex the root vertex of the tree; (3) $\forall w \in V_t$, suppose the indegree of w in G $deg^{-}(w) = d_i$, and outdegree of w in G $deg^{+}(w) = d_o$, w satisfies the following conditions: (i) $\exists v \in V_t$ s.t. $(v, w) \in E_t$; (ii) if $d_o = 0$, w is a leaf vertex of T, otherwise w is a non-leaf vertex of T; (iii) there is $d_i - 1$ dummy-copies of w as the dummy leaves of T.

Although vertex w has d-1 dummy vertices, PathGraph does not store d copies of the value of vertex w, but store only one copy of the value of w in the actual implementation. Dummy vertices do not introduce storage overhead for vertex values to the persistent storage.

Definition 2.4 (Reverse-Edge Traversal Tree): A reverse-edge traversal tree of G, denoted by $R = (V_r, E_r)$, is defined as follows: 1) $V_r \subseteq V$ and $E_r = E$; (2) There is one vertex, say $v_{rt} \in V_r$, $deg^{+}(v_{rt}) = 0$. We call this vertex the root vertex of the tree; (3) $\forall w \in V_r$, $deg^{-}(w) = d_i$, $deg^{+}(w) = d_o$, w satisfies the following conditions: (i) $\exists v \in V_t$ s.t. $(w, v) \in E_t$; (ii) if $d_i = 0$, w is a leaf vertex of T, otherwise w is a non-leaf vertex of T; (iii) there is $d_o - 1$ dummy-copies of w as the dummy leaves of T.

Different from a general traversal tree in which vertices of a graph are visited only once, in an edge traversal tree, each edge of a graph is visited once. Note that the edge traversal tree in PathGraph, strictly speaking, is a tree-like subgraph given that it contains dummy vertices, which are multiple appearances of a vertex. The reason that we keep the $r-1$ dummy leaf vertices for each of those vertices, say w, which has $r \geq 2$ in-degree or out-degree, is to model each of multiple paths reachable to/from w from/to the root vertex v_{root} independently for fast iterative computation. For simplicity, we call them **forward tree** and **reverse tree** respectively because the edges are all directed from or towards the root.

We can construct edge traversal trees using breadth-first search (BFS) or depth-first search (DFS). For instance, we can construct a forward-edge traversal tree $T = (V_t, E_t)$ of graph G by starting with adding a vertex v into V_t. We call v the root of T and put v into queue. Then we perform the following steps iteratively: $v = dequeue()$, $\forall e : v \rightarrow u \in E$, add e into E_t, remove e from G, $enque(u)$. This process iterates until no edges can be added from G. Fig. 1 gives illustrative examples with Fig. 1(b) as a forward tree and Fig. 1(c) as a reverse tree of the example graph in Fig. 1(a).

B. *Highlights of PathGraph Solution*

Many iterative graph algorithms explore the graph step-by-step following the graph traversal paths. The iterative process has to wade through many paths to find ones of interest. Thus, PathGraph presents a path centric graph computation model in which the input to PathGraph algorithms is a set of paths. However, PathGraph will provide application programmable interface (API) to support the vertex-centric programming model and our PathGraph engine will generate executable code that performs the graph analysis algorithm using the path centric model. More importantly, our PathGraph enabled graph processing system explores the path-centric model at both storage tier and computation tier.

At the computation tier, PathGraph first performs the path-centric graph partitioning to obtain path-partitions. Then it provides two principal methods for expressing graph computations: path-centric scatter and path-centric gather, both take a set of paths from some edge traversal trees and produce a set of partially updated vertices local to the input set of paths.

When the graph is large and cannot fit into the available memory for an iterative computation algorithm, we need to access graph data from disk. Existing vertex-centric or edge-centric approaches store a large graph data by partitioning the graph into a set of shards (partition chunks) with each storing vertices and their forward edges like in X-Stream [23] or with each storing vertices with their reverse edges like GraphChi [11]. Thus, when uploading a shard to memory, quite number of vertices and their edges are not utilized in the computation, leading to poor access locality and a waste of resource. In PathGraph, we improve the access locality by taking a path-centric approach at the storage tier as well. At storage tier, we divide a large graph by edge-traversal trees and divide large edge-traversal trees into paths. Then we cluster and store the paths of the same edge-traversal tree together while balancing the data size of each path-partition chunk. This path-centric storage layout can significantly improve the access locality because most of the iterative graph computations traverse along paths.

In the next two subsequent sections we will provide detailed description on path-locality based graph computation and path-locality based storage management.

III. Path-centric Graph Computation

Our path-centric graph computation model consists of algorithms for path-centric graph partitioning and path centric iterative graph computation.

A. *Locality based Graph Partitioning*

Given that most of the iterative graph computation algorithms need to traverse the paths where a vertex resides, an intuitive and yet effective solution is to partition a large graph by employing a path centric graph-partitioning algorithm to improve access locality. The path centric partitioning of a graph results in multiple path-partitions, each preserves the traversal path structure of the graph.

In order to effectively generate path partitions for a given graph, we use edge-traversal trees as the basic partitioning unit. By the edge-traversal tree definition given in the previous section, let $G = (V, E)$ be a graphs and $P_1, P_2, \ldots, P_k$ be a set of edge-traversal (sub) trees of G, where $P_i = (V_{p_i}, E_{p_i})$, $E_{p_1} \cup E_{p_2} \cup ... \cup E_{p_k} = E$, $E_{p_i} \cap E_{p_j}=\varnothing$; $\forall i \neq j$. We also call $P_1, P_2, \ldots, P_k$ $(1 \leq k \leq |E|)$ the k partitions of E. If vertex u is an internal vertex of P_i $(1 \leq i \leq k)$ then u only appears in P_i. If vertex v belongs to more than one partition, then v is a boundary vertex.

Before partitioning a graph, we first construct the forward-edge traversal trees of the graph using BFS as outlined in Section II-A. Now we examine each of the edge traversal trees to determine whether we need to further partition it into multiple smaller edge-traversal sub trees or paths (Algorithm 1). Concretely, we first examine a partition containing those vertices with zero incoming edges and perform the DFS on the tree. If the size of traversal tree exceeds the system-supplied limit for a partition, we split the edge traversal tree into multiple subtrees or paths. The iterative algorithm repeatedly operates on the edge traversal trees (partitions) until all edges are visited.

One way to define the partition size is by the number of edges hosted at each partition. For example, suppose that we set the $maxsize$ of a partition by 9 edges, then one way to partition the graph in Fig. 1(a) is to divide it into two partitions shown in Fig. 1(b). The left dotted green circle shows the first partition with four internal vertices and five boundary vertices and the right dotted red circle shows the second partition with four internal vertices and five boundary vertices. v_3, v_4, v_6, v_8, v_{10} are boundary vertices for both partitions. In PathGraph, each boundary vertex has a home partition where its vertex state information is hosted. In order to maximize the access locality, a boundary vertex shares the same partition as the one where the source vertices of its in-edges (reverse edges) belong. In the situation where the source vertices of its in-edges belongs to more than one partitions, the partition that has the highest number of such source vertices will be chosen as the home partition of this border vertex. If the numbers are same, the partition where the root of edge traversal tree is in will be chosen as home partition of the vertex. For the example of Fig. 1(b), The home partition for all boundary vertices v_3, v_4, v_6, v_8, v_{10} is P_1.

Using edge traversal trees as the partitioning units has a number of advantages. First, edge traversal trees preserve

Algorithm 1 Partition Graph Using Forward Trees

```
Input: partition $p$, vertex $v$
 1: if ($v == null$) then
 2:     set $v$ be a unvisited vertex with zero in-degree;
 3: end if
 4: Add all direct successors of $v$ into $set$;
 5: for each vertex $j \in set$ do
 6:     if ($p == null || no\ enough\ space\ in\ p$) then
 7:         Allocate a new partition $p$;
 8:     end if
 9:     add $edge : v \to j$ into $p$;
10: end for
11: for each vertex $j \in set$ do
12:     Partition-Graph-Using-Forward-Trees($p$, $j$);
13: end for
```

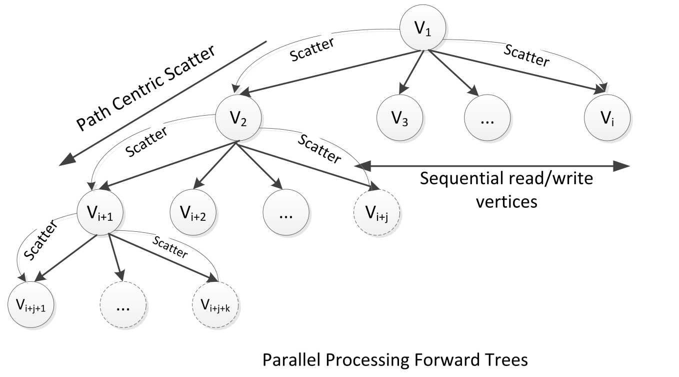
traversal structure of the graph. Second, most iterative graph algorithms perform iterative computations by following the traversal paths. Third, the edge-traversal tree based partition enables us to carry out the iterative computation for the internal vertices of the trees independently. This approach also reduces interactions among partitions since each partition is more cohesive and independent of other partitions.

B. Computing Model

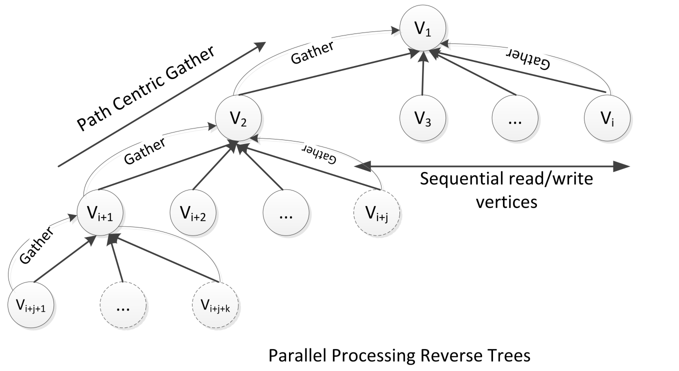
In PathGraph, the path centric computation model first performs the path-centric graph partitioning to obtain path-partitions. Then it provides two principal methods to perform graph computations: path-centric scatter and path-centric gather. Both take a path partition as an input and produce a set of partially updated vertex states local to the input partition.

Given an iterative graph analysis algorithm, at an iteration step i, the path-centric scatter takes a vertex of a forward tree (suppose it is v_2) and scatters the current value of v_2 to all of the destination vertices of v_2's forward-edges (Fig. 2(a)). Then the successors $v_{i+1}, ..., v_{i+j}$ of v_2 will initiate scatter operation and distribute theirs values to theirs successors. Different from scatter, the path-centric gather takes a vertex (suppose it is v_{i+1}) of a reverse tree and collects the values of source vertices v_{i+j+1},...,v_{i+j+k} of v_{i+1} to update its value (Fig. 2(b)). Similarly, v_2 of v_{i+1} will initiate gather and collect the values of its predecessors to update the value of v_2. For each partition, path-centric scatter or path-centric gather reads its vertex set, streams in paths, and produces an output stream of updates. The main loop alternately runs through an iterator over vertices that need to scatter state or over those that need to gather state.

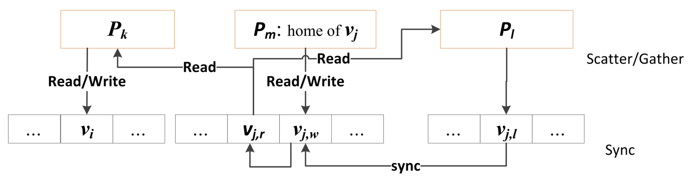
The scatter-gather process is followed by a synchronization phase to ensure all vertices at the ith iteration have completed the scatter or gather completely (Fig. 2(c)). For each internal vertex (also its dummy vertices) in a partition (eg. v_i in Fig. 2(c)), the engine can access its vertex value without conflict. But for a boundary vertex (eg. v_j in Fig. 2(c)) in a partition, its dummy vertices may distribute into several partitions. Given that the edges incidence with the dummy vertex v_j may be placed in several partitions. This may cause conflict when multiple partitions concurrently access the state of vertex v_j. In order to reduce conflict, like GraphLab [14], PathGraph creates two versions of state of v_j during the computation in memory: one version for other partitions to read and another



(a) Path Centric Scatter



(b) Path Centric Gather



(c) Sync

Fig. 2. Path Centric Computing Model

for the owner partition to read and write. Each non-owner partition will also maintains one version of vertex state for write. Considering v_j, $v_{j,r}$, $v_{j,w}$ reside in its owner partition is the primary copy of a vertex value. The computation in the owner partition can read and write $v_{j,w}$. Other partitions only access $v_{j,r}$. If partition P_l needs to update the value of v_j, it will write a temporary local copy of vertex value $v_{j,l}$. When a synchronization step is started, the local copies have to be propagated to the primary copy after each iteration. Then the primary copies have to be propagated to the read-only copy. The memory overhead for maintaining several versions of the state of v_j is introduced only during the execution of graph algorithms in memory. Upon completion, only one copy for each vertex is stored into persistent storage.

This scatter/gather-synchronization process is structured as a loop iteratively over all input paths until the computation converges locally over the input set of paths, namely some application-specific termination criterion is met. The computing model is illustrated in Fig. 2.

Comparing our path-centric approach in PathGraph with the vertex-centric and edge-centric approaches in the literature, we see a number of advantages of using path-centric computation model. For vertex-centric or edge centric computation model, the graph iterative computation is performed only at vertex or edge level and sync globally when each iteration step is completed for all vertices of the whole graph. In contrast, the path centric computation model performs iterative graph computation at two levels: at each path partition level within a graph and at each vertex level within a path partition. This allows parallelism at both path partition level and vertex level and also significantly improves the convergence speed of iterative graph computation by using parallelizing local sync for multiple path-partitions instead of performing the global synchronization at each iterative step.

IV. Path-Centric Compact Storage

To further improve the access locality for path-centric computation, we introduce a path-centric compact storage design, which includes the compact storage and vertex based indexing for edge-traversal tree based edge chunks. We argue that designing an efficient storage structure for storing big graphs in terms of their access locality based partitions is paramount for good performance in iterative graph computation. Such performance gain is multiplexed as the number of iterations increases. Thus, in PathGraph, we design the storage structure with two objectives in mind: improving storage compactness and improving access locality.

Conceptually, we represent each partition using an adjacency matrix. The adjacency matrix of a graph $G = (V, E)$ on n vertices is the $n \times n$ matrix where cell a_{ij} ($1 \leq i \leq n$, $1 \leq j \leq n$) is a bit to denote the presence ('1') or absence ('0') of an edge from vertex i to vertex j. Rows and columns of the matrix are vertex IDs sorted by their lexical order. We can then represent each row of the adjacent matrix by an vertex id as the key and followed by the adjacency set of this vertex and each element in the adjacency set of vertex v is a vertex id, which records the id of the corresponding neighbor vertex of v. Put differently, we can represent each edge of the graph using row id and column id that correspond to the '1' entry in the adjacency matrix. For each row, the edges share the same source vertex, namely row id. For each column, the edges share the same destination vertex, namely column id.

The matrix is huge and very sparse. In PathGraph, we design a compact storage format for internal representation of the matrix. The matrix is stored in two buckets, the forward bucket storing forward edge traversal trees and the reverse bucket storing reverse edge traversal trees. Fig. 3 gives a sketch of the storage layout.

A. Tree Partition based Storage Structure

Most of the existing graph processing systems use a vertex-centric model to store a large graph such that each vertex is stored together with either its out-edges or its in-edges. However, vertices on the same traversal paths may be stored in different disk pages and thus this vertex-centric storage model offers poor access locality for iterative graph computation.

To address this problem, in PathGraph, our idea is to store each edge traversal tree edge by edge in the depth-first traversal (DFS) order. The reason of using DFS instead of BFS is that vertices of a path in the DFS order are generally accessed subsequently in most of the iterative computations. For example, considering Fig.1(b) again, to store P_1 in DFS order, we begin at node v_1, and get a DFS ordered list of vertices, v_1 v_2 v_5 v_7 v_{10} v_6 v_8 v_3 v_4, among which v_8, v_3, v_4, v_6, v_{10} are boundary vertices with P_1 as their home partition. If a vertex does not have any child vertex in a partition (v_{10}, v_3, v_4 in P_1), we remove it from the list when storing the list. Thus, it is sufficient to only store the adjacent forward-edge set of v_1, v_2, v_5, v_7, v_6, v_8 consecutively in chunks of P_1. Similarly, P_2 consists of three traversal trees anchored at the boundary vertices v_8, v_3 or v_4 respectively. No matter which of the three boundary vertices is chosen by the algorithm to visit first, the algorithm will produce multiple DFS lists. Suppose one DFS list of P_2 is as follows: v_8 v_{11} v_{13} v_{14} v_3 v_{12} v_{14} v_6 v_{10} v_4 v_{13} v_8. The final list to be stored in chunks of P_2 is v_8 v_{11} v_3 v_{12} v_{14} v_4 v_{13}. We will explain the detailed structure of chunk in Section IV-B.

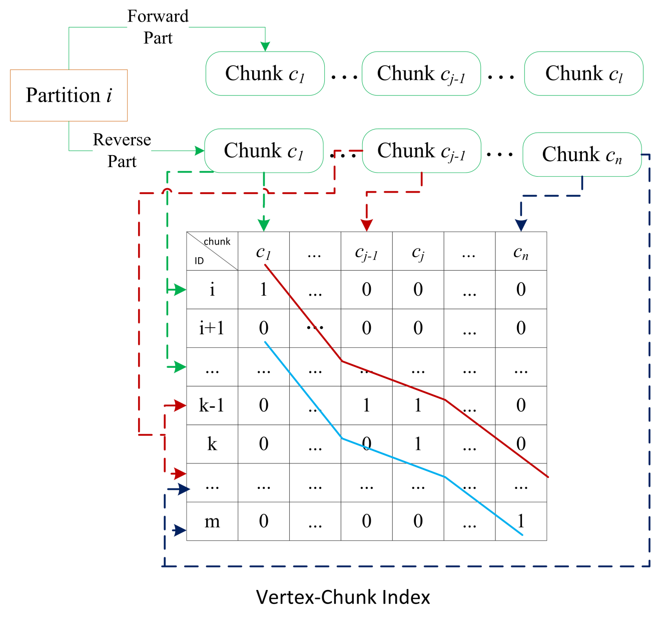
Given that PathGraph is by design a general purpose graph processing system, we support both forward-edge traversal tree based partition and also reverse-edge traversal tree based partition. This allows those graph computations that rely on destination vertex driven computation to be supported efficiently using our path-centric computation and storage models. Thus, PathGraph stores the matrix for each graph partition physically in two types of chunks: forward-edge chunks in forward order and reverse-edge chunks in reverse order, as shown in Fig. 3). Similarly, we remove those vertices without forward neighbors from the list for compact storage of the adjacency set in the reverse-edge chunks of the partition.

B. Chunk Storage Structure

Edges in each bucket are stored in fixed-size chunks. Since PathGraph stores edges in both forward bucket and reverse bucket, thus we maintain two types of chunks: forward chunks for forward trees and reverse chunks for reverse trees. To provide access locality, we assign chunks in each bucket to consecutive chunk IDs such that edges of a forward-edge (or reverse-edge) traversal tree are stored in physically adjacent chunks on disk as shown in Fig. 3(b). The size of chunk can be set according to the scale of data set and the memory capacity, IO page size.

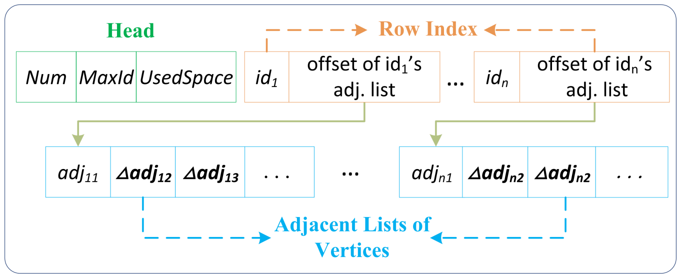
1) Data Structure: For the adjacency matrix of a graph, a chunk stores one or more rows of the matrix and each row is defined by vertex ID and its adjacent forward-edge set. We maintain the metadata and row index in the head of each chunk. In the head of each chunk, we store the number of rows, the maximal vertex ID in each chunk, the amount of used space, and the index of the rows stored in this chunk as shown in Fig. 3(b). The row index stores two pieces of information for each row: the row id and the offset of its adjacent set in the chunk. Since the chunk size is not big, storing the offset requires not much space. Also because row index indicates the minimal row id, it is not necessary to store $MinID$ again in the head of a chunk.

Adjacency set compression. The adjacent set for each row is sorted lexicographically by the IDs of the vertices in the adjacent set. We can further compress this adjacent set by utilizing the numerical closeness of vertex IDs. For example, the collation order causes neighboring edges to be very similar, namely the increases in vertex IDs of the adjacent set may often be very small. This observation naturally leads to a compression scheme for adjacency sets. Instead of storing the vertex ID, we only store the changes between each of the vertex IDs and the first id in the adjacency set.



chunk / ID	c_1	...	c_{j-1}	c_j	...	c_n
i	1	...	0	0	...	0
i+1	0	...	0	0	...	0
...	...	...	...	...	...	...
k-1	0	...	1	1	...	0
k	0	...	0	1	...	0
...	...	...	...	...	...	...
m	0	...	0	0	...	1

(a) Partition



(b) Chunk

Fig. 3. The Storage Scheme

***ID* compression.** *ID* (including delta values) is an integer. The size for storing an integer is typically a word. Modern computers usually have a word size of 32 bits or 64 bits. Not all integers require the whole word to store them. It is wasteful to store values with a large number of bytes when a small number of bytes are sufficient. Thus, we encode the *ID* using variable-size integer [34] such that the minimum number of bytes is used to encode this integer. Furthermore, the most significant bit of each compressed byte is used to indicate different IDs and the remaining 7 bits are used to store the value. For example, considering P_1 shown in Fig. 1(b), the adjacency set of v_1 is v_2, v_3, v_4. Supposing theirs ids are 2, 3, 400 respectively, the adjacency set of v_1 is stored as ”00000010 10000001 00000011 00001110”. In the byte sequence, the first byte is the id of v_2. 110001110 (after removing each byte's most significant bit of the third and fourth byte) is the difference of 400 (v_4) minus 2 (v_2).

C. Compact Indexing

One way to speed up the processing of graph computations is to provide efficient lookup of chunks of interest by vertex ID. We design the Vertex-Chunk index for maximizing the access locality. **Vertex-Chunk index** is created as an Vertex-Chunk matrix and it captures the storage relationship between vertex (rows) and chunks (columns) (Fig. 3). An entry in the Vertex-Chunk matrix denotes the presence ('1') or absence ('0') of a vertex in the corresponding chunk.

In each Vertex-Chunk matrix, rows and columns are sorted in an ascending order of IDs and sorted chunks respectively. Given that vertex IDs are generated by their visited order, and each chunk contains a set of row IDs, non-zero entries in the Vertex-Chunk matrix are around the main diagonal. We can draw one finite sequence of line segments, which go through the non-zero entries of the matrix. There are multiple curve fitting methods, such as lines, polynomial curves or Bspline. Complicated fitting methods involve large overhead when computing index. Thus, in the first prototype of PathGraph, we divide the vertex set into several parts (e.g., 4 parts). Since non-zero entries of the Vertex-Chunk matrix are expressed using one set of lines, we only need to store the parameters of one set of lines.

The Vertex-Chunk index gives the chunk ID ranger for each vertex. Instead of a full scan over all chunks, PathGraph only scans contiguous chunks which are in the ranger. Concretely, the system first hashes the vertex to get the corresponding chunk ID using the Vertex-Chunk index, then it examines the minimal ID and *MaxID* of each chunk and excludes those irrelevant chunks. For those candidate chunks whose minimal ID and *MaxID* contains this vertex, a binary search, instead of a full scan of all edges in each chunk will be performed.

D. Fast Vertex/Edge Lookup in Chunk

The compact storage and index structure in PathGraph enables fast lookup of a vertex and its adjacency set over a large graph stored on disk. For example, assume that PathGraph gets an edge with value (x, y). By using the x, we locate the corresponding chunks using index (see Section IV-C). We then binary search row index and get the row corresponding to x and its $offset$. Thus, we locate the adjacency set of row x using the $offset$. Now we need to search the adjacency set to find the edge using a binary search. Starting at the middle byte of the adjacency set, assume that it is ”10001001”, the PathGraph performs two steps to get the second element of edges. First, PathGraph reads the previous bytes and next bytes until the most significant bit of the bytes are not '1'. Then, PathGraph compares the input y (plus the id of the first vertex of the row) to the second element *ID* of the edge. If yes, it ends the process. Otherwise, it continues to compare and determine whether the input y is less or greater than the stored element *ID*. Depending on the input y, PathGraph repeats this process but only searches the top or bottom subset of the row and can quickly locate the edge (x, y).

V. Putting All Together

We implement the PathGraph system by combining path-centric storage and path-centric computation. Our first prototype is built by extending TripleBit [34]. Recall Section III-B, our path-centric computation performs iterative computation for each traversal tree partition independently and then performs partition-level merge by examining boundary vertices. Our model is mainly consisted of two functions: Gather (Algorithm 2) and Scatter (Algorithm 3). By introducing path-centric graph computation, we enable partition-level parallelism for graph processing.

Concretely, the Gather function (Algorithm 2) sweeps each

Algorithm 2 Gather

```
Input: $i$
1: for each chunk $c$ of reverse part of partition $i$ do
2:     read the row index $I$ of $c$;
3:     for each row $r$ of $I$ do
4:         read the $offset$ of $r$;
5:         read the adjacency set $s$ of $r$ using $offset$;
6:         read the vertex values of all vertex in $s$;
7:         update the vertex of $r$;
8:     end for
9: end for
```

chunk of the reverse part of a partition. For each chunk, the Gather function initially reads row index of the chunk. According to the *offset* corresponding to each row, the Gather function gets the adjacency set of this row, accesses all the vertex values in the adjacency set and then update the current vertex value by a user-defined merge function obtained from the API of PathGraph. The Scatter function sweeps each chunk of the forward part of a partition and scatter the value of a vertex to all its direct successors. Since trees are stored in DFS order, Gather and Scatter actually streams along the paths of a graph. Using PageRank as an example, after the Gather function processes a row of chunk, it will move to the next row of the chunk. It goes up in traversal tree level from *i*+1 to *i* since the vertex ids corresponding to rows are stored in DFS order. Thus, the Gather function increases locality.

We implemented and evaluated a wide range of applications, such as PageRank, SpMV, Connected Components and BFS, in order to demonstrate that PathGraph can be used for problems in many domains. Due to the length limit for paper, we will only describe how to implement PageRank algorithm in PathGraph. Algorithm 4 shows PageRank algorithm implemented in PathGraph. In this algorithm, each partition is processed in parallel. In each partition, Gather function is called to sweep the chunks in reverse part of each partition sequentially. After a Gather iteration finishes its execution, Sync function is called. Sync function processes each partition in parallel: each home partition of boundary vertices will collect local copies of the boundary vertices from other partitions, update their primary copies and set read-only copies as new values. Since each boundary vertex has only one home partition, Sync function will not induce conflict. In the implementation of PageRank algorithm, PathGraph bypasses three phases used in graph processing systems, such as X-Stream, and directly apply Gather, with improved performance.

Algorithm 3 Scatter

```
Input: $i$
1: for each chunk $c$ of forward part of partition $i$ do
2:     read the row index $I$ of $c$;
3:     for each row $r$ of $I$ do
4:         read the $offset$ of $r$;
5:         read the adjacency set $s$ of $r$ using $offset$;
6:         update the vertex values of all vertices in $s$;
7:     end for
8: end for
```

Algorithm 4 PageRank

```
1: for each iteration do
2:     parfor each $p$ of Partitions do
3:         Gather($p$);
4:     end parfor
5:     sync;
6: end for
```

VI. Evaluation

We evaluate the effectiveness of our PathGraph by conducting extensive experimental evaluation to compare with X-Stream and GraphChi. One reason of why we choose GraphChi and X-Stream as the competitors is that GraphChi and X-Stream are known to show better performance than other systems [11, 23]. The second reason is that GraphChi is based on vertex centric and X-Stream is based on edge centric approach. Both GraphChi and X-Stream are typical systems of the approaches. All experiments are conducted on a server with 4 Way 4-core 2.13GHz Intel Xeon CPU E7420, 55GB memory; CentOS 6.5 (2.6.32 kernel), 64GB disk swap space and one SAS local disk with 300GB 15000RPM. Before running all systems, we also compile all systems with gcc option O3 to be fair for such comparison. We used eight graphs of varying sizes to conduct evaluation: Amazon-2008, dblp-2011, enwiki-2013, twitter-2010, uk-2007-05, uk-union-2006-06-2007-05, webbase-2001 from [12] and Yahoo dataset [32]. Table I gives the characteristics of the datasets. These graphs vary not only in sizes, but also in average degrees and radius. For example, yahoo graph has a diameter of 928, and the diameter of enwiki is about 5.24 [12].

In the set of experiments reported in this paper, we constrain three systems under comparison to 50GB of maximum memory and use disk for secondary storage. The comparison includes both in-memory and out of core graphs. For example, X-Stream can load small data set (amazon, dblp, enwiki, twitter, webbase) into memory. For these datasets, X-Stream processes in-memory graphs. However, due to its large memory consumption, X-Stream cannot load uk2007, uk-union, and yahoo, into the memory. In latter cases, X-Stream processes out-of-core graphs on the testbed machine.

We select two traversal algorithms (BFS [33] and Connected Components) and two sparse matrix multiplication algorithms (PageRank and SpMV). Those graph algorithms are also used by GraphChi or X-Stream to highlight the best of their systems. We implement BFS, SpMV on GraphChi because GraphChi does not provide them in its source code.

A. Effectiveness of Path-Centric Storage

GraphChi stores edges in Compressed Sparse Row (CSR) format. It also stores the weights of edges, vertex values and their degrees. X-Stream stores edges only once and its basic storage consists of a vertex set, an edge list, and an update list. PathGraph stores edges in both forward chunks and reverse chunks though most of graph algorithms only need one of them. For example, PathGraph only uses reverse chunks when computing PageRank. Although all three systems have different storage structure for graphs, they all use matrix structure for edges and keep vertex state information and edge weights as attribute information.

TABLE I. DATASET CHARACTERISTICS

Data sets	Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
Vertices	735,322	986,286	4,206,757	41,652,229	105,896,268	133,633,040	118,142,121	1,413,511,394
Edges	5,158,012	6,707,236	101,355,853	1,468,365,167	3,738,733,633	5,507,679,822	1,019,903,024	6,636,600,779
Raw size	38.48MB	50.45MB	772.09MB	12.47GB	32.05GB	48.50GB	9.61GB	66.89GB

TABLE II. STORAGE IN MB

Data sets		Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
PathGraph	Forward part	13.9	18.0	260.1	3319.7	4867.3	7316.9	1877.6	15211.4
	Reverse part	14.2	18.0	219.9	3161.4	4603.2	6661.7	2022.7	10577.7
	Vertices, Degree	8.4	11.1	48.2	477.2	1204.6	1507.2	1343.5	8250.6
	Total	**36.5**	**47.1**	**528.2**	**6958.2**	**10675.1**	**15485.8**	**5243.7**	**34039.7**
X-Stream	Edge	59.1	76.8	1161.1	16820.5	42828.2	63092.0	11683.3	76024.1
	Nodes	≥#nodes*8 Bytes (depending on graph algorithms)							
GraphChi	Edge	20.4	26.5	390.6	5718.8	14320.6	21102.4	3886.8	27099.4
	Vertices, Degrees, Edge data	28.1	36.9	434.7	6083.6	15405.6	22435.7	5146.9	41558.4
	Total	48.5	63.4	825.3	11802.4	29726.2	43538.1	9033.7	68657.8

Table II compares the required disk space of PathGraph with X-Stream and GraphChi. The total storage size of PathGraph (including transpose) is smaller than the storage size of GraphChi and X-Stream, though PathGraph stores two copies of edges, one in forward bucket and one in reverse bucket. X-Stream has the largest storage due to its primitive storage structure. The total storage of GraphChi is 1.3X-2.8X that of PathGraph's storage.

The reason that PathGraph is more efficient comparing to GrahphChi and X-Stream is due to its compact edge storage structure. The storage for edges in X-Stream is 2.1X-4.5X that of storage in PathGraph. However, the storage of GraphChi for edge relation is 0.72X-1.51X that of PathGraph's storage. This is because GraphChi stores one copy of edges. We also observe that the maximal storage of PathGraph with both forward and reverse parts is 0.34-0.7X of GraphChi's edges. It shows that our storage structure is much more compact.

The storage size of GraphChi for vertex and edge attributes is 3.3X-14.8X of that in PathGraph, because GraphChi allocates storage for edge weights while PathGraph does not store them. In X-Stream, the data structure for vertex set varies depending on different graph algorithms. For example, in BFS algorithm, the vertex structure includes two fields (8 bytes). In PageRank, the vertex structure includes three fields (12 bytes). However, X-Stream requires at least 8 bytes for each vertex.

Fig.4 shows the memory usage comparison of the three systems running PageRank on uk-union. X-Stream requires one vertex state buffer which contains the states of all vertices; 4-5 same size stream buffers. X-Stream will use the amount of memory as its configuration specifies and will not change during its running. X-Stream will occupy 50GB memory during its run. Thus, its memory usage curve is a straight line. The data structure of GraphChi also requires huge memory. For example, its vertex structure contains 64 bits pointer for each edge of shards. Thus, its memory size is about eight times more than the size of graph data on disk. However, different from X-Stream, the memory usage of GraphChi varies during its execution because its memory is dynamically allocated and freed. Fig.4 shows GraphChi exceeds the memory limit we set and uses more than 54GB memory. Among three systems, PathGraph requires about 18GB peak memory. And after it finishes the execution, the memory for holding intermediate results will be freed.

In summary, PathGraph consumes less storage than X-Stream and GraphChi do for all the real graph datasets tested

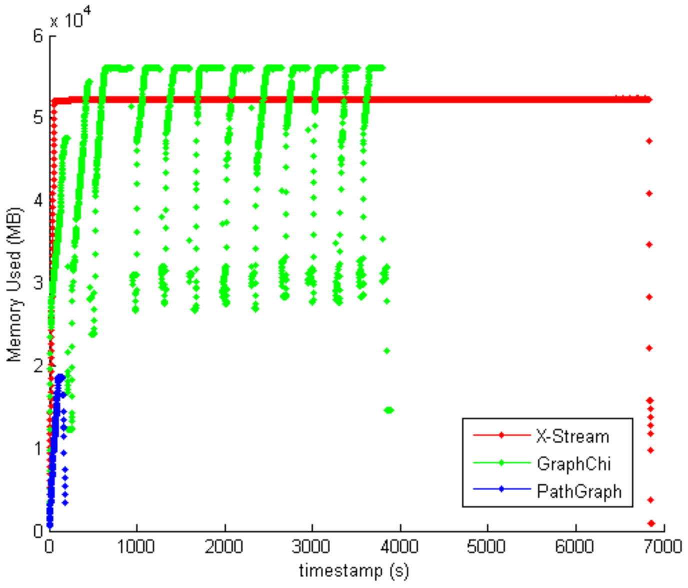


Fig. 4. Memory Usage when Running PageRank on uk-union

because PathGraph uses a compact storage structure, optimized for high access locality.

B. *Time Complexity for Loading Graphs*

Table III shows the time to import datasets into each of the three systems. The graph loading time refers to the built-time, basically consisting of the data traversal time and the time to write data into the storage. The first observation is that PathGraph imports data faster than X-Stream, but slower than GraphChi. When importing data into its storage, X-Stream just writes edges into its edge lists. GraphChi will write one copy of edges [11]. Both systems do not sort data. In contrast, PathGraph traverses the data using BFS and then writes data into a temporary buffer before it writes data into persistent storage. The workload of PathGraph to import data is heavier than X-Stream's importing workload. Even though, PathGraph is 1.4X-5.6X faster than that of X-Stream. The reason is that X-Stream uses a thread to execute the import task, but PathGraph and GraphChi uses 16 threads to do the task. Although the time of PathGraph importing data is 1.5X-

TABLE III. LOADING TIME (TIME IN SECONDS)

Data sets	Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
PathGraph	24	28	579	3875	7802	13656	1561	24863
X-Stream	46	58	787	11768	29243	43406	8694	60113
GraphChi	**4**	**6**	**89**	**1602**	**4098**	**6465**	**1053**	**8805**

TABLE IV. PAGERANK (TIME IN SECONDS)

Data sets	Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
PathGraph	**0.716**	**0.876**	**9.043**	**117.113**	**121.681**	**195.531**	**53.881**	**374.654**
X-Stream	1.452	1.822	26.116	1833.1	4763.82	6859.49	247.551	8497.93
GraphChi (out-of-core)	7.318	10.111	123.675	3388.11	2453.55	3951.75	892.604	12500.4
GraphChi (in-memory)	2.643	4.041	41.704	not available			399.241	not available

6.5X that of GraphChi's importing time, it brings significantly more benefits (better locality and performance etc.) for iterative graph computations.

C. PageRank

Table IV shows the execution time of running PageRank on eight graphs. PathGraph outperforms GraphChi and X-Stream. We include the performance of GraphChi's in-memory engine in our experimental comparison to PathGraph, which processes graphs in memory. Yahoo webgraph has a diameter much larger than other comparable graphs in the eight graph datasets tested. A high diameter results in graphs with a long traversal path structure, which causes X-Stream to execute a very large number of scatter-gather iterations, each of which requires streaming the entire edge list but doing little work. In comparison, PathGraph orders edges using BFS. PathGraph actually stores the shortest path between vertices with zero in-degree and other vertices, and it converges fast during PageRank computation.

X-Stream processes in-memory graphs faster than GraphChi. In out-of-core graphs, such as uk2007 and uk-union, GraphChi is faster than X-Stream. However, GraphChi is far slower than X-Stream in out-of-core graph yahoo. GraphChi's execution time on memory is less than half of that on magnetic disk.

To understand how the memory access pattern affects performance, we get the number of memory read/write and cache misses using Cachegrind [1]. Cachegrind can simulate memory, the first-level and last-level caches etc. Here, we only report the number of memory reads and writes, last-level cache read and write misses (LL misses row). The reason is that the last-level cache has the most influence on runtime, as it masks accesses to main memory [1]. We run experiments on 4 datasets because both GraphChi and X-Stream can not load larger datasets into memory. We also replace the storage structure of PathGraph with the similar storage structure as CSR or CSC used in GraphChi. We call the version of PathGraph as PathGraphCSR. PathGraphCSR stores trees as described in Section IV. The experimental results are shown in Table V. This allows us to understand the benefit of access locality independently from our storage structure and computing model.

The difference between path centric model and other models is that the CPU is able to do more work on data residing in the cache. We observed a significant reduction in cache miss of PathGraph. The typical factors (the cache misses of opponents divided by the cache misses of PathGraph) are among 2.8-13 (GraphChi/PathGraph), 2.3-24.9 (X-Stream/PathGraph) and 1.2-1.3 (PathGraphCSR/PathGraph). For PathGraphCSR, its LL cache misses are also smaller than the LL cache misses of the other two systems. Since PathGraphCSR shares the similar storage structure with GraphChi, the smaller cache misses of PathGraphCSR can contribute to our parallel computing model. We observed that GraphChi has the largest cache read miss and X-Stream has the largest cache write miss while both cache read miss and cache write miss of PathGraph are the smallest. This is because PathGraph has a regular memory access pattern while the memory access pattern of GraphChi and X-Stream could be random. One reason for small cache miss of X-Stream is that X-Stream combines nodes, degrees, matrix of graph into a single data structure while PathGraph and GraphChi separate graph data into different data structure.

We also observed much more saving on memory references, in which the vertex centric model is more than 1.1-2.17 times larger than our model and the edge centric model is 3.5-22.7 times larger than our model. This is because the storage structure of PathGraph is more compact than CSR used in GraphChi and PathGraphCSR and native storage of X-Stream. X-Stream has the largest memory references. The reasons are that its storage is native and the three phases of its graph computation requires many memory references. Although the memory references of GraphChi are far less than that of X-Stream, X-Stream is still faster than GraphChi. The reason is that last-level cache has the most influence on runtime.

Switching from the CSR storage structure to our storage structure, we see similar reduction for PageRank, but for different reasons. We reduce about 19%-24% of LL cache misses and 19%-27% of memory references for PageRank on 4 graphs. This is because our storage structure is more regular. CSR can also only sequentially search the data while our storage structure can perform binary search in the chunk.

D. SpMV, Connected Components and BFS

The experimental results are shown in Table VI. The first observation is that PathGraph performs much more efficiently than X-Stream and GraphChi on all algorithms over all graph datasets.

When executing SpMV, PathGraph improves X-Stream by nearly a factor of 2.1-28.4, improves GraphChi by a factor of 2.4-16. For twitter, uk2007, uk-union and yahoo, PathGraph is significantly faster than these two systems (The factors are more than 14). One reason is that those graphs are much bigger than Amazon and DBLP etc, and thus they require much more computing workload. Another reason is that GraphChi and X-

TABLE V. MEMORY READ/WRITE AND CACHE MISS

Data sets		Amazon		DBLP		enwiki		webbase	
		Read	Write	Read	Write	Read	Write	Read	Write
PathGraph	mem. refs	**325,068,748**	**67,413,223**	**420,310,473**	**86,283,931**	**5,952,627,465**	**1,255,625,657**	**59,712,508,254**	**11,242,856,082**
	LL misses	**1,355,255**	**516,537**	**1,806,381**	**688,966**	**145,119,289**	**5,134,542**	**253,567,011**	**72,181,698**
PathGraph-CSR	mem. refs	420,001,396	120,834,733	542,839,709	154,669,628	7,409,252,728	1,531,800,282	76,879,743,103	20,490,978,720
	LL misses	1,733,022	570,235	2,375,377	736,843	190,140,339	6,784,194	318,473,215	83,256,895
GraphChi (in-memory)	mem. refs	642,401,662	209,622,677	810,975,478	261,689,457	6,201,543,908	1,674,127,967	84,936,733,309	23,793,075,681
	LL misses	19,702,217	4,753,012	26,510,357	6,507,206	331,735,964	82,483,788	2,999,477,472	741,954,191
X-Stream	mem. refs	8,758,604,005	155,590,792	11,028,245,311	202,321,456	22,191,880,756	2,972,662,257	1,209,959,609,960	47,368,254,124
	LL misses	12,673,249	5,153,627	16,850,956	6,726,011	242,056,283	97,740,493	5,149,318,368	2,966,473,860

TABLE VI. SPMV, CONNECTED COMPONENTS (CC), BFS (TIME IN SECONDS)

Data sets		Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
SpMV	PathGraph	**0.249**	**0.739**	**3.449**	**40.443**	**61.665**	**82.552**	**30.097**	**215.579**
	X-Stream	1.231	1.535	24.132	661.955	1701.06	2348.03	217.385	2999.52
	GraphChi	1.337	1.797	25.369	643.394	904.357	1308.03	196.105	3502.62
CC	PathGraph	**1.181**	**2.114**	**14.022**	**208.263**	**224.951**	**265.791**	**132.52**	**2224.34**
	X-Stream	2.281	3.39	30.274	10311.4	$>$ 1 day	$>$ 1 day	637.478	$>$ 1day
	GraphChi	4.505	5.546	60.862	2918.94	8925.36	20385.6	568.747	13196.6
BFS	PathGraph	**1.138**	**1.214**	**5.231**	**148.955**	**195.426**	**234.775**	**81.237**	**738.718**
	X-Stream	1.664	1.781	28.452	3408.54	57538.7	7220.732	291.469	2871.583
	GraphChi	1.386	1.769	25.693	522.203	782.491	1200.24	192.89	3252.86

Stream can not load them into memory completely.

With connected components (CC), PathGraph offers the highest performance. It outperforms GraphChi by factors of 2.6 (dblp)-76.7 (uk-union). On larger datasets (uk2007, uk-union and yahoo), X-Stream runs more than one day and does not output results yet. Thus, we terminate its execution. For those datasets X-Stream runs CC successfully, our system typically outperform X-Stream with factors of 1.9-49.5.

For BFS, PathGraph outperforms X-Stream by factors of 1.5-294 and outperforms GraphChi by factors of 1.2-5.1.

E. Discussion

We attribute PathGraphs shorter runtimes to three factors.

Path-centric approach. The approach is consisted of path-centric computation and the storage structure. However, vertex-centric or edge centric approach does not care about storage structure. With our approach, not only computation but accessing data is more sequential than vertex-centric approach used in GraphChi and edge-centric approach used in X-Stream. Furthermore, different from three phases used in X-Stream, our approach only needs two phases: gather or scatter and sync. Thus, our approach also saves time.

Compact design in storage. The edges of graphs is most used during graph computation. The storage size of X-Stream and GraphChi for edges is larger than PathGraph's storage size for single copy of edges. Compact storage allows more efficient data access than X-Stream and GraphChi. For example, both systems require more time for I/O and place heavier burden on I/O. Although X-Stream scans edges sequentially, it reads source vertices at random. If the memory can not contain all intermediate results, X-Stream need write data into disk. It will induce much I/O.

Furthermore, our compact storage design also leads to efficient memory utilization. PathGraph can load all datasets in the experiment into memory, by virtue of the fact that its storage is more compact. However, it is difficult for X-Stream and GraphChi to do that. For example, X-Stream has one vertex state buffer and four stream buffers and produces updates for each edge. Each update information is generally stored as 8 bytes. Thus, X-Stream will require at least $8|E|$ bytes storage for intermediate results. GraphChi needs many shards, because it also needs to fit all edges of a shard into memory. This leads to more fragmented reads and writes that are distributed over many shards.

Less interactions among partitions. Our traversal tree partitioning approach will lead to less interaction among partitions when parallel executing graph algorithms. GraphChi and X-Stream do not consider relationships among vertices when they partition graph. Their approach will induce much communication among partitions.

VII. RELATED WORK

A fair amount of work has been engaged in graph data processing. Vertex centric, edge centric and storage centric approach are the three most popular alternative solutions for storing and processing graph data.

In vertex-centric computation model, computation is defined as kernels that run on each vertex in parallel. Many abstractions based on vertex centric model have been proposed [6, 11, 14, 15, 27, 29]. For instance, Pregel is a bulk synchronous message passing abstraction where vertices communicate through messages. GraphLab [14] also employs a vertex-centric model. GraphLab [14] is a sequential shared memory abstraction where each vertex can read and write to data on adjacent vertices and edges. However, different from the BSP model in Pregel, GraphLab allows asynchronous iterative computation. Since vertex centric access is random, GraphChi [11] breaks large graphs into small parts, and uses a parallel sliding windows method to improve random access on vertex. Xie etc. proposes a block-oriented computation model, in which computation is iterated over blocks of highly connected nodes instead of one vertex [31]. Tian etc [28] proposed a "think like a graph" programming paradigm, in which the partition structure is opened up to the programmers so that programmers can bypass the heavy message passing or scheduling machinery.

X-Stream [23] uses an edge-centric graph computation model. Comparing with vertex centric view, edge centric access is more sequential although edge traversal still produces an access pattern that is random and unpredictable.

Furthermore, executing algorithms that follow vertices or edges inevitably results in random access to the storage medium for the graph and this can often be the determinant of performance, regardless of the algorithmic complexity or runtime efficiency of the actual algorithm in use.

A storage-centric approach adopts optimized storage of graph. Common used storage structures for graph structure are adjacency matrix and the Compressed Sparse Row (CSR) format [11, 21], which is equivalent to storing the graph as adjacency sets. The research on graph databases proposed some more complicate storage structures. Many graph databases have been developed to manage, store and query large persistent graphs (e.g., DEX [16], Neo4j [2], HypergraphDB [9]). General graph databases are distinct from specialized graph databases such as triple stores and network databases. Particularly, in the context of direct labeled graph several engines such as YARS2 [8], RDF-3X [18–20], SHARD [22], SpiderStore [5], HexaStore [30], gStore [36], C-Store [3], TripleBit [34] et al. have been defined to store and process RDF graphs. Commonly, these engines provide APIs that facilitate the execution of graph tasks. They usually rely on physical structures and indices to speed up the execution of graph traversals and retrievals. However, the majority of them focus on individual triples. They do not provide powerful computational capabilities for iterative computation [11]. Thus, the implementation of algorithms for computing graph traversal patterns, object connectedness and shortest path lengths can be time-inefficient because of the huge number of potential self-joins required to execute these graph-based tasks.

A required step before processing graph is graph partitioning while maintaining good locality [35]. Graph partitioning problem has received lots of attentions over the past decade in high performance computing [4, 10, 13, 17, 24–26]. The most commonly used strategies in large-scale graph processing systems are vertex-centric hash partitioning. However, this approach has extremely poor locality [7], and incurs large amount communication across partitions in each iteration.

VIII. Conclusions

We have presented PathGraph, a path-centric approach for fast iterative graph computations on extremely large graphs. Our approach implements the path-centric abstraction at both storage tier and computation tier. In the storage tier, we follow storage-centric view and design a compact and efficient storage structure in order to accelerate data access. In computation tier, the path based parallel graph computation is used for promoting locality-optimized processing of very large graphs. Compared to both vertex-centric model and edge-centric model, our path centric approach takes full advantage of access locality in iterative graph computation algorithms. We have demonstrated that the PathGraph approach outperforms the state of art edge centric system – X-Stream and the vertex-centric representative – GraphChi on real graphs of varying sizes for a variety of iterative graph algorithms.

Acknowledgments

The research is supported by National High Technology Research and Development Program of China (863 Program) under grant No.2012AA011003. Ling Liu and Kisung Lee acknowledge the partial support by the National Science Foundation under grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing.

References

[1] Cachegrind. http://www.valgrind.org/, 2014.

[2] Neo4j. www.neo4j.org/, 2014.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. of VLDB 2007*, pages 411–422. ACM, 2007.

[4] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proc. of IPDPS 2006*, pages 124–124. IEEE Computer Society, 2006.

[5] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. Spiderstore: Exploiting main memory for efficient RDF graph representation and fast querying. In *Proc. of Workshop on Semantic Data Management (SemData@VLDB) 2010*, 2010.

[6] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of EuroSys'12*, pages 85–98. ACM, 2012.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30. USENIX, 2012.

[8] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *Proc. of ISWC/ASWC2007*, pages 211–224. Springer-Verlag Berlin, Heidelberg, 2007.

[9] B. Iordanov. Hypergraphdb: A generalized graph database. In *Proceedings of WAIM Workshops*, pages 25–36, 2010.

[10] G. Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *Proc. of Supercomputing 2003*, pages 1–11. ACM, 2003.

[11] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of OSDI'12*, pages 31–46. USENIX, 2012.

[12] Laboratory for Web Algorithmics (LAW). Datasets. http://law.di.unimi.it/datasets.php, 2013.

[13] K. Lee and L. Liu. Efficient data partitioning model for heterogeneous graphs in the cloud. In *Proc. of International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 25–36. ACM, 2013.

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

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD'10*, pages 135–146. ACM, 2010.

[16] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, M.-A. S.-M. J. Nin, and J.-L. Larriba-Pey. Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of CIKM 2007*, pages 573–582. ACM, 2007.

[17] I. Moulitsas and G. Karypis. Architecture aware partitioning algorithms. In *Proc. of ICA3PP 2008*, pages 42–53. Springer, Berlin, 2008.

[18] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proc. of SIGMOD 2009*, pages 627–639. ACM, 2009.

[19] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[20] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1-2):256–263, 2010.

[21] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceeding of SC'10*, pages 1–11. IEEE Computer Society, 2010.

[22] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Proc. of International Workshop on Programming Support Innovations for Emerging Distributed Applications 2010 (PSI EtA '10)*. ACM, 2010.

[23] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of SOSP'13*, pages 472–488. ACM, 2013.

[24] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[25] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *Proc. of IEEE/ACM International Conference on Computer Aided Design*, pages 726–733. IEEE Computer Society/ACM, 2003.

[26] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. *IEEE Transactions on CAD*, (3):504–517, 2006.

[27] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of SIGMOD'13*, pages 505–516. ACM, 2013.

[28] Y. Tian, A. Balminx, S. A. Corsten, S. Tatikonday, and J. McPhersony. From 'think like a vertex' to 'think like a graph'. *PVLDB*, 7(3):193–204, 2014.

[29] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *Proceedings of CIDR'13*, 2013.

[30] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[31] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14):2014–2025, 2013.

[32] YAHOO! Lab. Datasets. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g, 2013.

[33] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proc. of SC'05*. IEEE Computer Society, 2005.

[34] P. Yuan, P. Liu, B. Wu, L. Liu, H. Jin, and W. Zhang. TripleBit: a fast and compact system for large scale RDF data. *PVLDB*, 6(7):517–528, 2013.

[35] H. Zou, K. Schwan, M. Slawn̂ska, M. Wolf, G. Eisenhauer, F. Zheng, J. Dayal, J. Logan, Q. Liu, S. Klasky, T. Bode, M. Clark, and M. Kinsey. Flexquery: An online query system for interactive remote visual data exploration at large scale. In *Proc. of Cluster'13*. IEEE Computer Society, 2005.

[36] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *PVLDB*, (8):482–493, 2011.