PathGraph: A Path Centric Graph Processing System

Journal:	*Transactions on Parallel and Distributed Systems*
Manuscript ID	TPDS-2015-05-0341.R1
Manuscript Type:	Regular
Keywords:	D.1.3 Concurrent Programming < D.1 Programming Techniques < D Software/Software Engineering, E.1.d Graphs and networks < E.1 Data Structures < E Data, E.2 Data Storage Representations < E Data, F.1.2.d Parallelism and concurrency < F.1.2 Modes of Computation < F.1 Computation by Abstract Devices < F Theory of Computation, G.2.2.a Graph algorithms < G.2.2 Graph Theory < G.2 Discrete Mathematics < G Mathematics of Computing

Summary of Differences

In the SC 2014 paper, we reported the first version of the PathGraph system, which is implemented using Cilk+. In the manuscript, we report the second generation of our graph processing system, PathGraph, which is built on top of PathGraph-baseline but with a number of significant and original developments tuned for both in-memory (see Table 5, 6 of the revised manuscript) as well as out-of-core cases (see Table 8 of the revised manuscript). The major differences are as follows:

(1) We improve the memory and disk access locality for iterative computation algorithms on large graphs by modeling a large graph using a collection of tree-based partitions. For each tree partition, we re-label vertices using DFS or BFS in order to preserve consistency between the order of vertex ids and vertex order in the paths (Section 2, 3). The goal of our approach mainly cares about load balance and locality, not communications. In Table 3, 4, we show the loading time and data balance of our partitioning approach.

(2) We discuss the complexity of our computing model (Scatter/gather) in Section 4.1 by comparing it with the computing models of vertex centric based and edge centric based graph processing engines. We also demonstrate how Gather and Scatter are used in two graph algorithms: BFS and SpMV in Section 7.

(3) To provide well balanced workloads among parallel threads at tree partition level or chunk level, we introduce multiple stealing points based task queue to allow work stealing from multiple points in the task queue. In the experiments, we show that our scheduling approach achieves better load balance than the state art of technology - Cilk+ (see Table 9 Load balance).

(4) We also demonstrate the performance (Table 3, 5, 6, 7, 8) and speedup (Fig. 6) of PathGraph by comparing it with the representative vertex centric based graph processing engine GraphChi, edge centric based engine X-Stream, and PathGraph-baseline. Our experimental results show that PathGraph outperform the systems for its performance.

(5) We also rewrite most sections (Section 1, 2, 3, 4, 7 etc.) to improve the clarity and highlights of our system.

PathGraph: A Path Centric Graph Processing System

Pingpeng Yuan, Changfeng Xie, Ling Liu, *Fellow, IEEE,* and Hai Jin, *Senior Member, IEEE*

Abstract—Large scale iterative graph computation presents an interesting systems challenge due to two well known problems: (1) the lack of access locality and (2) the lack of storage efficiency. This paper presents PathGraph, a system for improving iterative graph computation on graphs with billions of edges. First, we improve the memory and disk access locality for iterative computation algorithms on large graphs by modeling a large graph using a collection of tree-based partitions. This enables us to use path-centric computation rather than vertex-centric or edge-centric computation. For each tree partition, we re-label vertices using DFS in order to preserve consistency between the order of vertex ids and vertex order in the paths. Second, a compact storage that is optimized for iterative graph parallel computation is developed in the PathGraph system. Concretely, we employ delta-compression and store tree-based partitions in a DFS order. By clustering highly correlated paths together as tree based partitions, we maximize sequential access and minimize random access on storage media. Third but not the least, our path-centric computation model is implemented using a scatter/gather programming model. We parallel the iterative computation at partition tree level and perform sequential local updates for vertices in each tree partition to improve the convergence speed. To provide well balanced workloads among parallel threads at tree partition level, we introduce the concept of multiple stealing points based task queue to allow work stealings from multiple points in the task queue. We evaluate the effectiveness of our PathGraph approach by comparing with recent representative graph processing systems such as GraphChi and X-Stream. Our experimental results 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.

Index Terms—Graphs and networks, Concurrent Programming, Graph algorithms, Data Storage Representations.

1 Introduction

We are entering the big data era. A large number of information contents generated from business and science applications are highly connected datasets, which fuels a growing number of large scale graph analysis applications [10]. Most of existing graph processing systems address the iterative graph computation problem by programming and executing graph computation using either vertex centric or edge centric approaches, which suffer from a number of problems, such as the lack of access locality and the lack of efficient partitioning method.

Access Locality. Most of the iterative graph computation algorithms need to traverse the graph from each vertex to learn about the influence of this vertex over other vertices in the graph through its connected paths. Also graphs can be irregular and unstructured. Thus, vertices or edges that are accessed together during iterative computations may be stored far apart, leading to high cache miss rate and poor access locality at both memory and secondary storage. This is primarily due to two factors: First, the storage structure for graphs typically fails to capture the topology based correlation among vertices and edges. Furthermore, existing graph partitioning approaches, such as random edge partitioning or vertex based hash partitioning tend to break connected components of a graph into many small disconnected parts, which leads to extremely poor locality [9], and causes large amount communication across partitions during each iteration of an iterative graph computation algorithm. Thus, the runtime is often dominated by the CPU waits for memory access.

Efficient Graph Partitioning and Processing. Achieving high performance in processing a large scale graph requires efficient partitioning of the graph such that the graph computation can be distributed among and performed by multiple parallel threads or processes. Existing graph processing systems are primarily based on the vertex centric or edge centric computation models, and the random hash based partitioning of a graph by vertices or edges. However, random hash partitioning often produces ”balanced” partitions in terms of vertices or edges but such balanced partitions result in disconnected components of the original graph, which leads to poor access locality and consequently causes imbalanced computational workload. We argue that one of the key challenges for improving the runtime of iterative graph computations is to develop effective mechanisms for load balance among parallel tasks. A popular way to manage load balance is to use a parallel task scheduling method that supports work-stealing based on a non-blocking data structures that prevents contention during concurrent operations [3]. An open problem for work stealing is the contention among two or more thieves when they choose the same victim, causing access conflicts and imbalance loads.

To address these two problems, we argue that many iterative graph algorithms explore the graph step-by-step following the graph traversal paths. For each vertex to be processed, we need to examine all its outgoing or incoming edges and all of its neighbor vertices. 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. The iterative computation of a graph has to wade through many paths and then converges when all vertices have completed

- *P. Yuan, C. Xie, H. Jin are with the School of Computer Sci. & Tech., Huazhong Univ. of Sci. & Tech., Wuhan, China, 430074. e-mail:ppyuan@mail.hust.edu.cn, hjin@mail.hust.edu.cn,*
- *L. Liu is with the College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332. email:lingliu@cc.gatech.edu*

their iterative computation. Thus, it motivates us to partition a graph into a collection of tree based partitions, each consisting of multiple traversal paths. The second motivation is to support access locality through both a compact storage for graph data and a storage placement method using tree-based partitions instead of random collections of edges or vertices.

Following the motivations, we develop a path-centric graph processing system − PathGraph for fast iterative computation on large graphs with billions of edges. Concretely, PathGraph presents a path centric graph computation model from two perspectives: First, the input to the PathGraph algorithms is a set of paths, even though PathGraph provides API to support the vertex-centric programming model, our PathGraph engine will generate executable code that performs the graph analysis algorithm using the path centric computation model. Second, the graph processing system of PathGraph 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 edge traversal trees and produce a set of partially updated vertices local to the input set of paths. The path-centric scatter/gather model allows PathGraph to parallels the iterative computation at tree partition level and performs sequential local updates for vertices in each tree partition to improve the convergence speed.

At storage tier, 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 contrast, existing vertex-centric or edge-centric approaches partition and store a graph into a set of shards (partition chunks) with each shard storing vertices and their outgoing (forward) edges like in X-Stream [21] or vertices with their incoming (reverse) edges like GraphChi [12]. Thus, when uploading a shard to memory, it is possible that some vertices and their edges are not utilized in the computation, which leads to ineffective data access or poor access locality. In addition to model a large graph using a collection of tree-based partitions for improving the memory and disk locality for iterative computation algorithms, we also design a compact storage using delta-compression, and store a graph by tree-based partitions in a DFS order. By clustering highly correlated paths together, we further maximize sequential access and minimize random access on storage media.

To evaluate the effectiveness of our path-centric approach, we compare PathGraph with two recent representative graph processing systems: GraphChi and X-Stream, and show that the path-centric approach outperforms the vertex-centric system GraphChi and the edge-centric system X-Stream on a number of graph algorithms for both in-memory and out-of-core graphs.

2 System Overview

In this section, we give an overview of the PathGraph system from three perspectives. First, we present the basic concepts used in PathGraph. Second, we present the system architecture of PathGraph, which outlines the main components of PathGraph. We will present the technical description of the core components of PathGraph in the subsequent sections.

2.1 Preliminary

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, we will present our approach in the context of directed connected graphs in the rest of the paper.

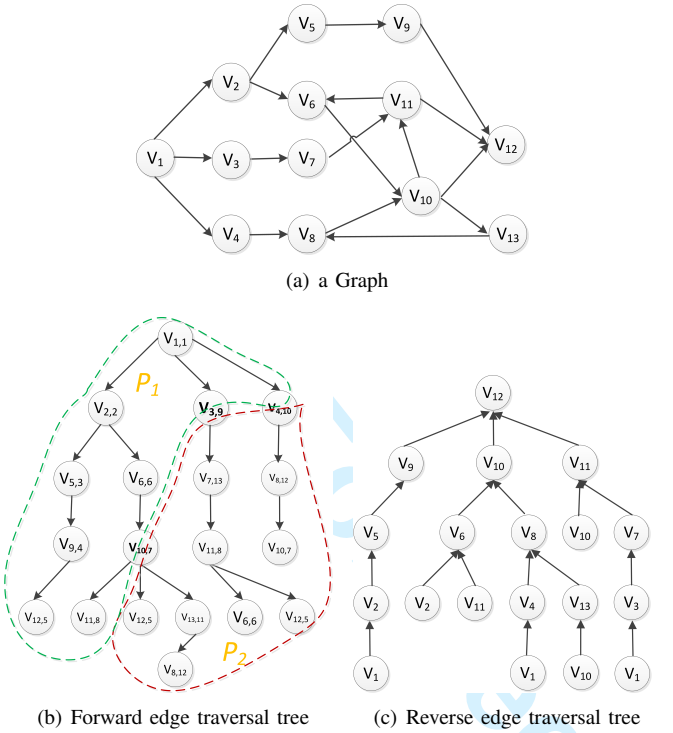
Definition 2.1 (Graph). *A graph $G = (V, E)$ is defined by a finite set V of vertices, $\{v_i | v_i \in V\}$ $(0 \leq i < |V|)$, and a set E of directed edges which connect certain pairs of vertices, and $\{e = (u, v) \in E | u, v \in V\}$. The edge (u, v) is referred to as a forward edge or out-edge of vertex u and a reverse edge or in-edge of vertex v.*

Definition 2.2 (Vertex Degree). *Let $G = (V, E)$ denote a graph. For any vertex $u \in V$, let IE(u)=$\{v | v \in V, (v, u) \in E\}$, and OE(u) = $\{w | w \in V, (u, w) \in E\}$. We say that the in-degree of u, denoted by $deg^-(u)$, is $\|IE(u)\|$ and the out-degree of u, denoted by $deg^+(u)$, is $\|OE(u)\|$.*

Definition 2.3 (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.*

There is a class of iterative graph computation algorithms that need to examine each vertex in a graph by traversal of the graph along the forward direction or the reverse direction. Consider PageRank, one way to compute the page rank is to traverse the graph by following the reverse edges. 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. Another way is to traverse the graph by following the forward edges. Its updated vertex state (rank score) will be scattered to all destination vertices of its out-edges for the PageRank computation in the next iteration. In each iteration, every vertex u will be examined for PageRank computation by examining its adjacent (neighbor) vertices, resulting in examining all vertices in a graph. We call such a reverse traversal tree rooted at u the reverse-edge traversal tree of u. Similarly, we can also define the forward-edge traversal tree of u. A graph may have one or more forward or reverse edge traversal trees. In the following, we will give the definition of forward or reverse edge traversal tree/forests.

Definition 2.4 (Forward-Edge Traversal Tree). *A forward-edge traversal tree/forest of G, denoted by $T_f = (V_f, E_f)$, is defined as follows: 1) $E_f = E$; (2) $\exists v_{rt} \in V_f$ such that $deg^-(v_{rt}) = 0$. We call this vertex the root vertex of the tree; (3) $\forall u \in V_f$, u satisfies the following conditions: (i) $\exists v \in V$, u is v or u is a dummy copy of v; (ii) $\exists v \in V_f$ and v is a child of u s.t. $(u, v) \in E_f$; (iii) if u is a leaf of T_f, u is a*



(a) a Graph (b) Forward edge traversal tree (c) Reverse edge traversal tree

Fig. 1. A graph and its Forward-edge Traversal Tree, Reverse-edge Traversal Tree

dummy vertex or $deg^+(u) = 0$. *if* u *is a non-leaf of* T_f, u *has at most* $deg^+(u)$ *children by its out-edges.*

Definition 2.5 (Reverse-Edge Traversal Tree/Forest)**.** *A reverse-edge traversal tree/forest of* G, *denoted by* $T_r = (V_r, E_r)$, *is defined as follows: 1)* $E_r = E$; *(2)* $\exists v_{rt} \in V_r$ *s.t.* $deg^+(v_{rt}) = 0$. *We call this vertex the root vertex of the tree* T_r; *(3)* $\forall u \in V_r$, u *satisfies the following conditions: (i)*$\exists v \in V$, u *is* v *or* u *is a dummy copy of* v; *(ii)* $\exists v \in V_r$ *and* v *is a child of* u *s.t.* $(v, u) \in E$; *(iii) if* u *is a leaf of* T_r, u *is a dummy vertex or* $deg^-(u) = 0$. *if* u *is a non-leaf of* T_r, u *has at most* $deg^-(u)$ *children by its in-edges.*

A traversal forest contains traversal trees, each of which is a connected component of the graph. In a traversal forest, each tree is a directed tree, namely it is a directed path from the root to any other vertex in the tree. For the forward traversal tree or forest T_f, there is $deg^-(u) - 1$ dummy-copies of u as the dummy leaves in T_f. For the reverse traversal tree/forest T_r, there is $deg^+(u) - 1$ dummy-copies of u as the dummy leaves in T_r. For simplicity, we call them the **forward tree** and the **reverse tree** respectively.

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 $T = (V_f, E_f)$ of graph G by starting with adding a source vertex v (in-degree is zero) into V_f. We call v the root of T and put v into list. Then we perform the following steps iteratively: remove an element v from the list, $\forall e : v \to u \in E$, add e into E_t, remove e from G, add u into the list. This process iterates until no edges can be added from G. Depending on DFS or BFS, we push and pop on the same side of list, namely push onto the right (top) of the queue and pop from the left (bottom) of the queue,

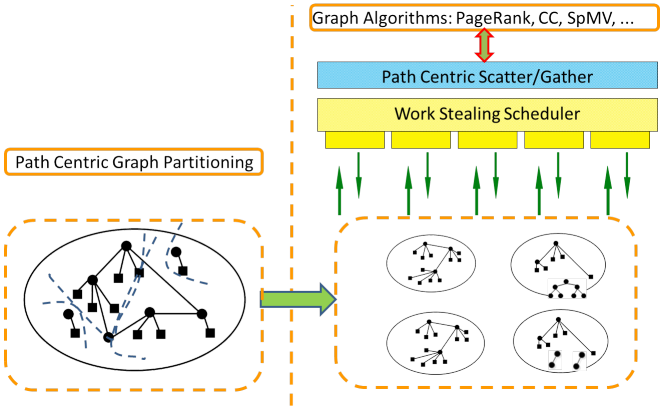


Fig. 2. The Architecture of PathGraph

to get a sequence of vertices from the queue. The process will visit each vertex and edge. Suppose there is at least one vertex v not visited and let w be the first unvisited vertex on the path from s to v. Because w was the first unvisited vertex on the path, there is a neighbor u that has been visited. But when we visited u we must have looked at edge(u, w). Therefore w must have been visited. For the example graph in Fig. 1(a), Fig. 1(b) shows a forward traversal tree of the graph and Fig. 1(c) shows a reverse traversal tree of the graph. Consider the example graph in Fig. 1(a), vertex v_1 has three forward-edges, thus, in the reverse-edge traversal tree shown in Fig. 1(c), v_1 has $deg^+(v_1) - 1$=2 dummy vertices.

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 $deg^+(w) - 1$ dummy leaf vertices for each of those vertices, say w, which has $deg^+(w) \geq 2$ or $deg^-(w) \geq 2$, is to model each of the multiple paths reachable to (from) w from (to) the root vertex v_{root} independently for fast iterative computation. Thus, given a forward traversal tree/forest $T_f = (V_f, E_f)$ of $G = (V, E)$, the size of V_f has the upper bound of $\sum_{u \in V} deg^-(u)$. Similarly, for a reverse traversal tree/forest $T_r = (V_r, E_r)$ of a graph G, the size of V_r has the upper bound of $\sum_{u \in V} deg^+(u)$.

Traversing a graph from a vertex u gives us a traversal tree rooted at u. Any vertex in the graph that is reachable from vertex u by following u's forward (reverse) edges is a part of the forward (reverse) tree of u. The part of a forward (reverse) tree is a sub-tree of the forward (reverse) tree.

Definition 2.6 (Sub-Tree)**.** *Let* $T = (V_T, E_T)$ *denote a forward or reverse edge traversal tree. A tree* $S = (V_S, E_S)$ *is a subtree of* T *if and only if (iff)* S *satisfies the following conditions: (i)* $V_S \subseteq V_T$, $E_S \subseteq E_T$, *(ii)* $\exists v_{rt} \in V_S$ *s.t.* $\forall (u, v_{rt}) \in E_T \to (u, v_{rt}) \notin E_S$. *We call* v_{rt} *the root vertex of* S, *and (iii)* $\forall y \in V_S$, *we have* $(v_{rt}, y) \in E_S$ *or* $\exists v_1, ..., v_k (k \geq 1)$ *such that* (v_{rt}, v_1),(v_1, v_2), ...,(v_{k-1}, v_k),$(v_k, y) \in E_S$.

2.2 System Architecture

The PathGraph system consists of five main functional components: (1) The Path Centric Graph Partitioner, which takes a raw graph G as input and transforms the graph G into a collection of forward-edge or reverse-edge traversal trees. (2)

The Path Centric Compact Storage module, which store the graph based on the traversal trees. For each traversal tree, we re-label the vertices to improve the access locality. (3) The Work Stealing schedule using Multi-ended Queue, which provides load balance among multiple partition-level parallel threads. (4) The Path Centric Graph Computation module, including path centric gather and scatter functions. (5) Graph algorithms, which include PageRank, BFS, CC, SpMV etc. Figure 2 shows an architectural sketch of PathGraph.

We will dedicate the subsequent four sections to path-centric graph partitioning, path-centric computation using scatter-gather, the workstealing scheduler and the path-centric compact storage respectively.

3 Path-Oriented Graph Partitioning

Since PathGraph runs on a machine, our partitioning approach differs from general partitioning approach and mainly cares about load balance and locality. Given that most of the iterative graph computation algorithms need to traverse the paths, an intuitive solution is to partition a large graph by employing a path centric graph-partitioning algorithm to improve access locality. Our path-oriented graph partitioning approach actually partitions the traversal trees of a graph as sub-trees, each of which preserves the traversal path structures of the graph. In many graph processing systems, vertices are stored in the lexical order of their vertex IDs assigned randomly. It induces worse locality. Thus we describe how we re-label the vertices in each traversal tree in order to improve the access locality.

3.1 Traversal Tree Based Graph Partitioning

In order to effectively generate path partitions for a given graph without breaking many paths, 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 $P_1, P_2, \ldots, P_k$ $(0 < k \leq |E|)$ the k partitions of E. If vertex u is an internal vertex of P_i $(0 < i \leq k)$ then u only appears in P_i. If u belongs to more than one partition, then u is a boundary vertex.

Before partitioning a graph, we first define the partition size is by the number of edges hosted at each partition. Then we construct the forward-edge traversal trees (or reverse-edge traversal trees) of the graph using BFS as outlined in Section 2. 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. Concretely, considering the forward trees, we first examine a partition containing those vertices with zero incoming edges, perform DFS or BFS on the tree and add the visited edges into the partition. If the edge number of the partition exceeds the system-supplied limit for a partition, we will construct another partition using the similar process. When partitioning edges, we can divide the edges of a vertex into multiple partitions or place all of them in a partition. The process repeatedly operates on the edge traversal trees (partitions) until all edges of a graph are visited.

For example, suppose that we set the $maxsize$ of a partition by 10 edges, then one way to partition the graph in Fig. 1(a) is to divide it into two partitions shown in Fig. 1(b). In Fig. 1(b), the edges of $v_{10,7}$ are divided into partition P_1, P_2. However, we can place the edges of $v_{10,7}$ into a partition. The former case is evener in edge distribution than the latter case. The left dotted green circle shows P_1 with four internal vertices and six boundary vertices and the right dotted red circle shows P_2 with three internal vertices and six boundary vertices.

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 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. Considering the example in Fig. 1(b), the home partition for all boundary vertices $v_{3,9}$, $v_{4,10}$, $v_{6,6}$, $v_{10,7}$, $v_{11,8}$, $v_{12,5}$ is P_1.

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

3.2 Path Preserving Vertex Relabeling

In PathGraph, all forward traversal trees, as well as reverse traversal trees are Direct Acyclic Graph (DAG) if we ignore dummy vertices. Thus, we can find a topological order of traversal trees in the graph using depth-first search. Given that original vertex IDs do not indicate the access order, this motivate us to improve locality using relabeling vertices within each traversal tree. In order to keep the topological order of all vertices, we improve locality by relabeling vertices within each traversal tree. For forward traversal tree, we must assign small ids to vertices that are in the front of list, and then bigger ids to vertices in the back of the list. However, for reverse traversal tree, we must assign bigger ids to vertices in the front of the list and smaller ids to vertices in the back of the list. Algorithm 1 is used to find a topological order of a graph and then re-label vertices according to their orders. Since dummy vertices are not real vertices (Step 6), we skip them when constructing topological order of a graph. Concretely, we visit vertices using DFS, then recursively assign all vertices unique id in the visiting order.

By Algorithm 1, all vertices are re-labeled for each traversal tree. Figure 1(b) shows that each vertex is associated with the original label (the first label of each vertex) and the new label (the second label of each vertex). For example, the original label of $v_{10,7}$ is 10 and its new label is 7. For v_1, its new label is the same as its original label, which is 1. Also the resulting traversal tree defines a partial order for vertex access.

Algorithm 1 Assigning IDs using DFS of tree

```
1: while there are vertices without assigning new ids do
2:     get an vertices v without new ids;
3:     DFS-AssigningIDs(v);
4:
5: function DFS-AssigningIDs(vertex v)
6: if (v is not a dummy vertex) then
7:     AssignID(v);
8:     for each vertex u: v → u do
9:         DFS-AssigningIDs(u);
```

Lemma 1. *Suppose $T = (V_T, E_T)$ is a forward (or reverse) edge traversal tree in $G = (V, E)$. $\mathcal{P}$ is the set of paths in T. $\forall p =< v_0, v_1, ..., v_k > \in \mathcal{P}$, where edge $e : v_i \rightarrow v_{i+1} (0 \leq i < k) \in E_T$. If v_i, v_{i+1} are not dummy vertices, then $id(v_{i+1}) > id(v_i)$ $(0 \leq i < j < k)$. And $\forall u, v \in V$, if $id_u > id_v$, then the vertex v should be visited before the vertex u.*

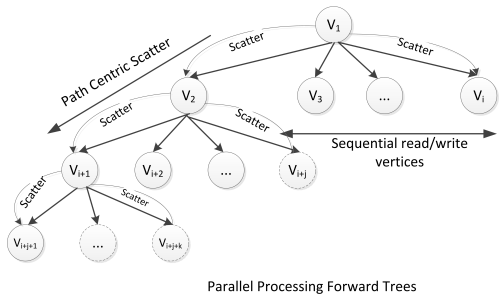
Lemma 2. *Let S be a subtree of the traversal tree $T = (V_T, E_T)$ and MinId and MaxId are the minimal and maximal id of non-dummy vertices in S. $\forall v \in V_T$, if v is not a dummy vertex of S, then $MinId \leq id(v) \leq MaxId$.*

This lemma states that the ids of non-dummy vertices in a subgraph S of a traversal tree T must fall within the range defined by its MinID and MaxID. Thus, any vertices with their vertex ids outside the range do not belong to S. The proof of these two lemmas is straightforward by using Algorithm 1 and the definition of MinID and MaxID.

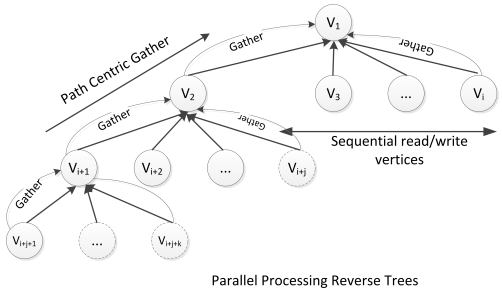
4 Path Centric Scatter-Gather

Scatter-gather programming model is sufficient for a variety of graph algorithms [12, 21]. In the model, developers define a scatter function to propagate vertex state to neighbors and a gather function to accumulate updates from neighbors to recompute the vertex state. 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 one or several traversal trees or sub-trees as an input and produce a set of partially updated vertex states local to the input partition.

Given an iterative graph 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. 3(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. 3(b)). Similarly, then 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



(a) Path Centric Scatter



(b) Path Centric Gather

Fig. 3. Path Centric Computing Model

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. For each internal vertex in a partition, the engine can access its vertex value without conflict. But for a boundary vertex in a partition, it may lead to conflict when multiple partitions access the vertex. In order to reduce conflict, like GraphLab [16], PathGraph stores two versions of boundary vertex values: one version for other partitions to read and one for the owner partition to read and 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 need 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.

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. 3.

4.1 Complexity Analysis

To simplify the complexity analysis, we assume the processor accesses one vertex or edge at a time, triggering the update function. This assumption does not limit expressiveness, since the resulting execution policy is at the granularity of vertices or edges whatever model is used. We can extend the discussion to sets of vertices or edges as well.

Suppose we are given a directed graph $G = (V, E)$. We denote the cost to write and read a value respectively by W,

R. In the edge-centric approach, The computation is structured as a loop, each iteration of which consists of a scatter phase followed by a gather phase [21]. Before gather phase, a shuffle phase is required to re-arrange updates such that each update appears in the update list of the streaming partition containing its destination vertex. Thus, the complexity to process a graph is $O(3\|E\|(W+R)+2\sum_{v\in V} deg^-(v)R)$.

In vertex-centric model, both the scatter and the gather phase iterate over all vertices. In each iteration, a function specified by programmers accesses and modifies the value of a vertex and its incident edges [12]. Thus, the total complexity is $O(\sum_{v\in V}((deg^+(v)+deg^-(v))(R+W)+R))$=$O(2\|E\|(W+R)+\|V\|R)$.

Since our approach is consisted of two operations (*scatter* and *gather*), theirs complexity is different: the complexity to execute *scatter* on the out-edges of vertex v is $deg^+(v)W+R$; the complexity of execute *gather* on in-edges of v is $deg^-(v)R+W$. Thus, the complexity to process graph G is $O(\sum_{v\in V}(deg^+(v)W+R))$ (scatter) or $O(\sum_{v\in V}(deg^-(v)R+W))$ (gather). The path centric computation model performs iterative graph computation at three levels: at each path partition level within a graph, at each chunk level within a path partition and at each vertex level within a chunk. A key benefit of computation on partition level, chunk level instead of vertex level is improved locality of reference: by loading a block of closely connected vertices into the cache, we reduce the cache miss rate.

5 Work Stealing Scheduler

Given that a graph is split into a collection of traversal trees by the traversal tree partitioning, the graph processing can be performed on each of these traversal trees separately and independently. However, the workload for processing different partitions can be different due to different numbers of vertices and number of edges in each partition and the amount of communication cost among partitions. One way to deal with such potential load imbalance is to devise a work stealing schedule. It is recognized that work-stealing needs to address several problems due to its asynchronous nature [3–6]. Work stealing is mainly performed using the double-ended queues, or so called *deques*, each of which is a sequence container that can be operated on either the front end or the back end of the queue. In a work-stealing schedule, multiple thieves can steal work from a owner. Thieves and owner operate on the different ends of a deque. Although the use of double ended queues indeed lowers the synchronization cost among local and stealing threads, it does not eliminate the conflicting problem between threads, because two or more threads may steal works from the same deque. We argue that the situation in which two or more steal operations are executed on the same deque with no synchronization among thieves should not be allowed. A thief must have some guarantee that other thieves concurrently steal work from the same queue will be blocked or forced to abort due to long wait. Our experimental results on the first version of the PathGraph system - PathGraph-baseline show that more than 50% of steal operations conflict. One obvious reason is that even multiple threads can steal work

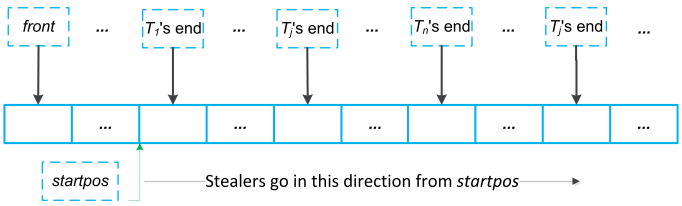


Fig. 4. Multiple ended queue

from one end of the deque, only one work stealing thread is active at any given time.

To improve work stealing performance, in PathGraph, we introduce multi-ended queues to replace dequeues (double-ended queues). Concretely, our task pool is a multiple-ended queue, denoted by *meque*. Each thread has a *meque* of which it is the owner. The tasks in the *meque* can be given to other work stealing threads by the work stealing scheduler. Each owner thread executes tasks by taking one task at a time from the front of queue. However, unlike deques that can only allow work stealing from the back end, *meque*s allow thieves steal works at multiple end points of the sequence container except the front end. Concretely, each thread is assigned an end for steal. Consider Figure 4, there are n threads: T_1, ..., T_n. Each *meque* has a start position for work stealing, denoted by $startpos$, and thieves can steal tasks from the end points after $startpos$ in order to avoid possible conflicts with the owner. From $startpos$, each thread will be assigned an end for future stealing. The end points assigned to the consecutive threads will be adjacent. For example, the ends for thread j and thread $j+1$ will be adjacent. If a thread needs to steal tasks from the *meque* again, a new end point will be assigned. considering the end point for the previous steal as pre_end. If thread j chooses thread i as a victim again, and there are still tasks in the *meque* of thread i available, thread j will be assign a new end: $(pre\_end+n-1)\ \%\ mq.length$ (Line 6) where $\%$ is the MOD operator. The process will repeat until all tasks in the *meque* are examined and completed.

Algorithm 2 steal

```
1:  choose a victim whose queue is not empty;
2:  get meque mq of the victim;
3:  no is the No. of the stealer.
4:  tail = mq.tail;
5:  if (pre_end ≥ startpos) then
6:      cur_end = (pre_end + n − 1) % mq.length;
7:  else
8:      cur_end = (startpos + no) % mq.length;
9:  if (cur_end > tail) then
10:     set cur_end as one value of [cur_end − no, tail];
11: suc=compare_and_swap(state[cur_end],1,0 );
12: if suc then
13:     pre_end = cur_end;
14: else
15:     abort;
16: execute mq[cur_end];
```

When a thread finds out that its *meque* is empty, it acts globally by stealing the task at the specified end of the *meque*s of a

victim. Our algorithm maintains an end for each thread that is never decremented. Each thief takes stolen elements from its assigned end of the *meque* by the work stealing scheduler. By utilizing multi-ended queues to replace double-ended queues, PathGraph eliminates the conflict between thieves when there are enough tasks. When there are not enough tasks in *meque*, PathGraph allows thieves to steal the same task. In the case, our approach can still avoid some conflicts because not all threads choose the same end points.

In our approach, we use a compare-and-swap (CAS) instruction to achieve synchronization (Line 11). It compares the contents of a memory location to a given value and modifies the content of that memory location to a given new value only if they are the same. This is done as a single atomic operation. Abort is returned if the thread fails to steal a task. Therefore, *meque* is more effective for concurrent work stealing and it allows threads to steal works more efficiently under certain circumstances, especially with very long sequences, where stealing work from only one end prohibits other stealing.

Not all threads will steal tasks from the *meque* because their own *meque*s are not yet empty or they choose other victims. Since the owner thread of the *meque* will proceed to take tasks from the front of its *meque*, the owner will sweep the *meque*. When it finds the un-stolen tasks, the un-stolen tasks will be reclaimed by the owner of the *meque*. The tasks can be inserted into the tail of the *meque* or executed by the owner.

6 Path-Centric Compact Storage

Simply storing the graph does not enable efficient access of graph data. 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. To see this, consider vertex v, its in-neighbor vertex set I and the out-neighbor vertex set O. Let $w \in I$ is updated, then 1) read the values of w and v; 2) compute the new value of v; 3) the modification is written to v. Similarly, when v is changed, its out-neighbors also do the same operations. According to the above analysis, the read and write operations are propagated along paths. In fact, in many algorithms, the value of a vertex only depends on its neighbors' values. Thus, if the neighbors are stored following paths, the system can sequentially access values of the neighbors, and the amount of random access could be reduced. Consider this, in PathGraph, we design the storage structure with two objectives in mind: improving both storage compactness and access locality.

To further improve the access locality for path-centric computation, we introduce a path-centric compact storage design (Fig. 5), which includes the compact storage for traversal trees and vertex based indexing for edge chunks.

6.1 Storing Tree Partitions

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 $v_{1,1}$, and get a DFS ordered list of vertices: $v_{1,1}$, $v_{2,2}$, $v_{5,3}$, $v_{9,4}$, $v_{12,5}$, $v_{6,6}$, $v_{10,7}$, $v_{11,8}$, $v_{3,9}$, $v_{4,10}$, among which $v_{3,9}$, $v_{4,10}$, $v_{6,6}$, $v_{10,7}$, $v_{11,8}$, $v_{12,5}$ are boundary vertices with P_1 as their home partition. If a vertex does not have any child vertex in a partition ($v_{3,9}$, $v_{4,10}$ in P_1), we remove it from the list when storing the list. Thus, the adjacent forward-edge set of $v_{1,1}$, $v_{2,2}$, $v_{5,3}$, $v_{9,4}$, $v_{6,6}$, $v_{10,7}$ are consecutively stored in chunks of P_1. Similarly, P_2 consists of three traversal trees anchored at the boundary vertices $v_{10,7}$, $v_{3,9}$, $v_{4,10}$ 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: $v_{10,7}$, $v_{12,5}$, $v_{13,11}$, $v_{8,12}$, $v_{3,9}$, $v_{7,13}$, $v_{13,8}$, $v_{6,6}$, $v_{12,5}$, $v_{4,10}$, $v_{8,12}$, $v_{10,7}$. The final list to be stored in chunks of P_2 is $v_{10,7}$, $v_{13,11}$, $v_{3,9}$, $v_{7,13}$, $v_{11,8}$, $v_{4,10}$ $v_{8,12}$.

Given that PathGraph is by design a general purpose graph processing system, we support both forward-edge traversal tree based partitions and reverse-edge traversal tree based partitions.

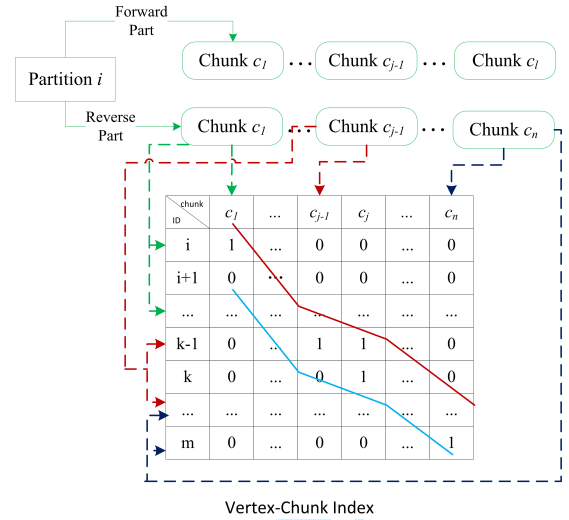
Since vertices are relabeled in their forward edge traversal tree or reverse edge traversal tree according to their topological sort order. Then vertices and their adjacency lists are stored in lexical order. Thus, vertex IDs are stored in an ascending order of IDs when we store DFS lists. This order guarantees the locality when access data. It is not possible to store both forward parts and reverse parts in DFS order while maintaining lexical order of IDs. However, most of graph algorithms do not need both forward part and reverse part equally. For example, PathGraph uses reverse parts when computing PageRank and BFS only accesses forward parts. Thus, we can store one part of a partition in both DFS order and lexical order. If forward buckets are stored in DFS order, then reverse buckets will be stored in lexical order. Otherwise, the reverse buckets are stored according to the order given by post-order depth-first traversal, and forward buckets are stored in lexical order.

6.2 Chunk Storage Structure

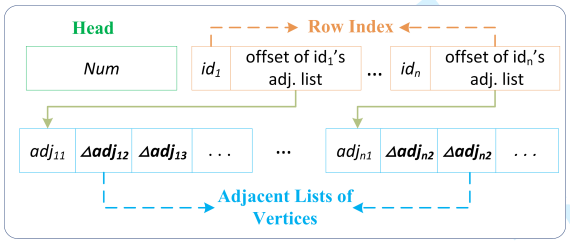
Edges in each bucket are stored in fixed-size chunks. To provide access locality, we assign chunks in each bucket to consecutive chunk IDs such that edges of a forward-edge (or reverse-edge) traversal tree are stored in physically adjacent chunks on disk as shown in Fig. 5(b).

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 set (Fig. 5(b)). In the head of each chunk, we store the number of rows stored in the chunk. The row index stores two pieces of information for each row: the row id and the offset of its adjacent set in the 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 vertex ids to be very similar, namely the increases in vertex IDs of the adjacent set may often be very small. Thus, instead of storing the vertex



(a) Partition



(b) Chunk

Fig. 5. The storage scheme

ID, we only store the changes between each of the vertex IDs and the first id in the adjacency set.

***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 [26] 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).

6.3 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 speeding up the lookup. **Vertex-Chunk index** is created as an Vertex-Chunk matrix and it captures the storage relationship between vertex (rows) and Chunks (columns) (Fig. 5(a)). An entry in the Vertex-Chunk matrix denotes the presence ('1') or absence ('0') of a vertex in the corresponding chunk. Since the adjacent sets are stored physically in two buckets, we maintain two Vertex-Chunk index structures: one for forward ordering chunks and the other for reverse ordering chunks.

In each Vertex-chunk matrix, rows and columns are sorted in an ascending order of IDs and sorted chunks respectively. Thus, non-zero entries in the Vertex-chunk matrix are around the main diagonal. We can draw two 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, etc. Complicated fitting methods involve large overhead when computing index. Thus, 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 possible "middle" Chunk ID for each vertex. Instead of a full scan over all chunks, PathGraph only scans contiguous chunks nearby the "middle" Chunk. For those candidate chunks, a binary search, instead of a full scan of all edges in each chunk will be performed.

7 Implementation and Applications

We implement PathGraph by combining path-centric storage and path-centric computation. Our first prototype is built by extending TripleBit [26]. Recall Section 4, our path-centric computation performs iterative computation for each traversal tree partition independently and then perform partition-level merge by examining boundary vertices. By introducing path-centric graph computation, we enable partition-level parallelism for graph processing. As described in Section 4, our model is mainly consisted of two functions: Gather and Scatter.

The two functions can process the units with different grain sizes. For example, Gather can process tree partitions (or subtrees) and chunks. Here, we give two implementations of Gather function. In order to distinguish them, we name the Gather function processing trees as Gather-T (Algorithm 3), Gather function on chunks as Gather-C (Algorithm 4). The Gather-T function executes operations user defines on each predecessors of a given vertex i. The Gather-C function sweeps each chunk of the reverse part of a partition. For each chunk, the Gather-C function initially reads row index of the chunk. According to the *offset* corresponding to each row, the Gather-C 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. Using PageRank as an example, after the Gather-C 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.

Algorithm 3 Gather-T

Input: vertex i

```
1: read the adjacency set s of i in reverse tree;
2: for each vertex v in s do
3:     execute user defined ops;
```

Algorithm 4 Gather-C

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 adjacency set s of r using offset;
5:         execute user defined ops;;
```

Algorithm 5 Scatter-T

Input: vertex i

```
1: read the adjacency set s of i in forward tree;
2: for each vertex v in s do
3:     execute user defined ops;
```

Here, we only give the implementation of Scatter function on trees (Algorithm 5). The Scatter-T (Algorithm 5) function reads adjacency set of a vertex in 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.

We implemented and evaluated a wide range of applications, such as PageRank, BFS, SpMV, and Connected Components, 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 SpMV algorithm and BFS using $Gather$ and $Scatter$ as an illustration.

In SpMV algorithm (Algorithm 6), each partition is processed in parallel. In each partition, Gather function is called to sweep the chunks in reverse part of each partition sequentially. In the implementation of SpMV algorithm, PathGraph bypasses three phases used in graph processing systems, such as X-Stream, and directly apply Gather, with improved performance.

Algorithm 6 SpMV

```
1: for each iteration do
2:     parfor root p of each Partitions do
3:         Gather-C(p);
4:     end parfor
```

We also implement a parallel 'level-synchronous' breadth-first search (BFS) algorithm [15] using $Scatter$. The $level$ is the distance the algorithm travels from the source vertex. The algorithm explores the neighbor nodes first, before moving to the next level neighbors. So global synchronization is needed at the end of each traversal step to guarantee this. The algorithm is shown in Algorithm 7. In the algorithm, $bag[d]$ is an unordered-set data structure which stores the set of vertices at distance d (layer d) from v_0. v_0 is the source vertex initiating BFS, and $v.dist$ is the distance from v_0 to v. Each iteration processes the vertexes in the layer d by calling the $Scatter$ function. The operations user defined in the Scatter function check all the neighbors of vertex v. For those that should be added to $bag[d+1]$, we update its distance from the source vertex and insert the vertex into the bag of $d+1$ layer (Line 14 - 17). In the end of each iteration, the sync function is called to wait for the layer is finished.

Algorithm 7 BFS

Input: v_0

```
1: parfor each vertex v do
2:     v.dist=∞;
3: end parfor
4: bag[0]=create-bag();insert-bag(bag[0],v0);
5: v0.dist=0; d=0;
6: while bag[d] IS NOT EMPTY do
7:     bag[d+1]=create-bag();
8:     parfor each v of bag[d] do
9:         Scatter-T(v);
10:        sync;
11:    end parfor
12:    d = d + 1;
13:
14: Function bfs-op(vertex v)
15: if v.dist == ∞ then
16:    v.dist=d+1;
17:    insert-bag(bag[d+1],v);
```

8 Evaluation

We implement the PathGraph system described in this paper, which incorporates the multi-ended queue based work stealing scheduler and vertex relabeling scheme to further provide the load balance among multiple partition threads and enhance access locality. Since the first version of PathGraph [27] is developed using Cilk plus (http://en.wikipedia.org/wiki/Cilk), we name the previous PathGraph as PathGraph-baseline. Both systems are implemented using C++, compiled with GCC.

We compare PathGraph against PathGraph-baseline. Our goal is to evaluate whether our implementation of work stealing with *meque*s is competitive with the state of the art technology. PathGraph and PathGraph-baseline share same storage structure. Thus, when comparing the storage of systems, we mention them as PathGraph. We also choose X-Stream and GraphChi as competitors since they showed better performance [12, 21] and they are typical systems of vertex centric and edge centric model.

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, and Yahoo dataset from [13]. Table 1 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 [13]. We select two traversal algorithms (BFS and Connected Components) and two sparse matrix multiplication algorithms (PageRank and SpMV). We implement BFS, SpMV on GraphChi since GraphChi does not provide them.

All experiments (excluding out-of-core case) 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. Since X-Stream and GraphChi perform better with larger memory, we constrain X-Stream and GraphChi to 50GB of maximum memory, which is close to physical

TABLE 1
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 2
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
	Vertices	≥#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

memory size. 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, X-Stream can not load uk2007, uk-union, and yahoo, into the memory. In latter cases, X-Stream processes out-of-core graphs on the testbed machine.

8.1 Effectiveness of Path-Centric Storage

Table 2 compares the required disk space of PathGraph with X-Stream and GraphChi. The total storage size of PathGraph (including forward part and reverse part) 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.

GraphChi stores edges in 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 part when computing PageRank.

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 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.

TABLE 3
Loading time (time in seconds)

Data sets	Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
PathGraph	29	35	592	4421	15479	27035	2893	39672
PathGraph-baseline	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**

8.2 Loading Graphs

The graph loading time basically consists of reading the original data, traversal time and the time to write data into the storage for future operations. Loading time is actually built-time. When importing data into its storage, X-Stream just writes edges into its edge lists. GraphChi will write one copy of edges [12]. Both systems do not sort data. In contrast, PathGraph traverses graphs, relabels vertices and then writes data into a temporary buffer before it writes data into persistent storage. For PathGraph and X-Stream, this process only needs to be done once per graph. However, GraphChi builds its storage according to different graph algorithms [21]. Here, we report the loading time of GraphChi for PageRank.

The first observation is that PathGraph imports data slower than GraphChi, PathGraph-baseline, but faster than X-Stream (Table 3). The reason why PathGraph imports data slower than PathGraph-baseline is that PathGraph needs to transfer the adjacency list to reverse adjacency list. The workload of PathGraph to import data is heavier than X-Stream's importing workload. Even though, PathGraph is 1.3X-3X faster than that of X-Stream. The time of PathGraph importing data is 2.7X-7X that of GraphChi's importing time. Considering that GraphChi will build its storage structure for each algorithm, the loading time of our systems is not big. Furthermore, although the loading workload of our system is heavy, it brings significantly more benefits (better locality and performance etc.) for iterative graph computations.

8.3 Data Balance

We compare our approach with METIS [17]. METIS is an edge-cut approach, but our approach is a vertex-cut approach. It is not convenient to compare the data skew of two approaches directly, such as the vertex or edge numbers of partitions. Thus, we compare METIS with our approach in

TABLE 4
Data balance of partitioning

		Max. Exec. Time	Min. Exec. Time	Average Exec. Time
Amazon	METIS	1.672	0.78	1.3835
	Our approach	1.416	0.954	1.1054
DBLP	METIS	3.001	0.423	1.8431
	Our approach	2.044	1.144	1.4586

terms of execution time. We cannot get METIS to work on larger datasets due to insufficient memory. Therefore we can only collect results over the amazon and DBLP dataset. We initially compute the k-way original partitions of a graph using METIS. We then add vertexes that are connected via an edge from any vertex within the original partition along with the edges to the partition. Finally, we load the partitions into PathGraph and get the running time on the partitions.In the experiment, our approach places all edges of a vertex into a partition instead of dividing them into several partitions (see Section 3.1). Table 4 shows the maximal/minimal computing time on partitions for METIS and our approach. In METIS, the maximal computing time is 2.14X (amazon)/7.09X (DBLP) of the minimal computing time. For our approach, the maximal computing time is 1.48X (amazon)/1.786X (DBLP) of the minimal computing time.

8.4 Scatter Operation

Here, we execute SpMV, Connected Components and BFS since the algorithms are implemented using scatter operation. The experimental results are shown in Table 5. The first observation is that PathGraph performs much more efficiently than the three systems on all algorithms over all graph datasets.

For BFS, PathGraph improves PathGraph-baseline by factors of 1.1-1.8. PathGraph outperforms X-Stream by factors of 1.9-334.6 and outperforms GraphChi by factors of 1.9-5.9.

When executing SpMV, PathGraph improves PathGraph-baseline by a factor of 1.11-1.31, X-Stream by nearly a factor of 2.7-31.8, improves GraphChi by a factor of 3.2-18.2. For twitter, uk2007, uk-union and yahoo, PathGraph is significantly faster than GraphChi and X-Stream (The factors are more than 15). One reason is that those graphs are much bigger than Amazon and DBLP etc, and thus they require much computing workload. Another reason is that GraphChi and X-Stream can not load them into memory completely.

As with CC, PathGraph offers the highest performance. It outperforms PathGraph-baseline by factors of 1.2-1.9, GraphChi by factors of 4.9 (dblp)-90.1 (uk-union). On larger datasets (uk2007 etc), X-Stream runs more than one day and does not output results yet. Thus, we terminate its execution. For those datasets X-Stream which runs CC successfully, our system typically outperform X-Stream with factors of 3-81.4.

8.5 Gather Operation

In the experiments, PageRank is implemented using Gather operation. Table 6 shows the execution time of running PageRank (4 iterations) on eight graphs. In our experimental comparison to PathGraph, we include the performance of GraphChi's in-memory engine, which processes graphs in memory. PathGraph improves PathGraph-baseline by factors of 1.1-1.9. PathGraph outperforms X-Stream by factors of 2.5-66. PathGraph outperforms GraphChi (out-of-core) and GraphChi (in-memory) by factors of 4.5-9.6.

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.

Memory references and cache miss. 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. However, we store trees as described in Section 6. We call the version of PathGraph as PathGraphCSR. The experimental results are shown in Table 7.

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 3.3-15 (GraphChi/PathGraph), 2.7-28.9 (X-Stream/PathGraph), 1.4-1.6 (PathGraphCSR/PathGraph) and 1.1-1.2(PathGraph-baseline/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. A cache write miss generally causes the least delay than cache read miss, because the write can be queued and there are few limitations on the execution of subsequent instructions. 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.

TABLE 5
SpMV, Connected Components (CC), BFS (time in seconds)

Data sets		Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
BFS	PathGraph	**0.617**	**0.916**	**4.634**	**113.581**	**171.971**	**203.301**	**65.452**	**583.066**
	PathGraph-baseline	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
SpMV	PathGraph	**0.208**	**0.565**	**3.103**	**36.216**	**55.614**	**73.758**	**23.154**	**192.341**
	PathGraph-baseline	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	**0.718**	**1.133**	**8.144**	**126.725**	**187.606**	**224.036**	**86.468**	**1823.09**
	PathGraph-baseline	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

TABLE 6
PageRank (time in seconds)

Data sets	Amazon	DBLP	enwiki	twitter	uk2007	uk-union	webbase	Yahoo
PathGraph	**0.591**	**0.683**	**7.667**	**102.442**	**74.518**	**103.554**	**41.401**	**330.138**
PathGraph-baseline	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	N/A			399.241	N/A

TABLE 7
Memory read/write and cache miss

Data sets		Amazon		DBLP		enwiki		webbase	
		Read	Write	Read	Write	Read	Write	Read	Write
PathGraph	mem. refs	**191,384,872**	**61,265,888**	**246,196,023**	**75,976,184**	**3,399,048,230**	**1,122,110,774**	**34,311,392,889**	**10,143,669,388**
	LL misses	**1,144,845**	**472,375**	**1,596,759**	**613,271**	**122,181,461**	**4,582,567**	**214,866,645**	**65,919,038**
PathGraph-baseline	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

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 5.6-35.3 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% more of LL cache misses and 19% more of memory references for PageRank on 4 graphs. This is because our storage structure is more regular. CSR can only sequentially search the data while our storage structure can perform binary search in the chunk.

8.6 Processing Out of Core Graphs

We now compare the out-of-core performance of PathGraph to X-Stream and GraphChi. We constrain four systems to 16GB of memory and using the SAS for storage. For the experiments, we choose yahoo data set and run the four algorithms on it because four systems can not load yahoo into memory. The results (Table 8) show PathGraph finishes its execution faster than the other systems.

TABLE 8
Processing out-of-core graphs (yahoo)

	PageRank	BFS	CC	SpMV
PathGraph	372.729	614.721	1915.679	211.396
PathGraph-baseline	426.325	771.829	2335.274	237.872
X-Stream	10167.47	3309.27	$>$ 1day	3403.29
GraphChi	16013.84	3839.07	17029.38	4125.62

We attribute PathGraph shorter runtimes to two factors. The first factor is the path-centric approach, in which many updates are absorbed by vertices in the same partitions. The second contributor is our small usage of available bandwidth from the disk. In order to fit the data that will be processed into memory, X-Stream and GraphChi need swap data in and out. This leads to more fragmented reads and writes. In contrast, PathGraph's storage is compact. Comparing to the other two systems, PathGraph requires less bandwidth. Moving further, PathGraph processes partitions in parallel, by virtue of the fact that each partition does not depend on each other much and it only needs to fit the vertex data for the partition into memory.

8.7 Load Balance of Scheduling and Speedup

Here, we report the load balance of our approach against Cilk plus. We use the same tasks generator, and the same compiler when running PathGraph-baseline and PathGraph. In Table

TABLE 9
Load balance (yahoo)

		Min. Exec. Time	Max. Exec. Time	Median Exec. Time	STDEV
PageRank	Cilk	0.888	1.336	0.9605	0.1203
	meq	0.94	1	0.9465	0.0147
BFS	Cilk	0.865	1.098	0.997	0.0710
	meq	0.996	1.001	0.998	0.0012
SpMV	Cilk	1	1.198	1.082	0.0694
	meq	0.969	1.025	0.995	0.0212
CC	Cilk	0.815	1.242	1.0115	0.1217
	meq	0.999	1.005	1.001	0.0013

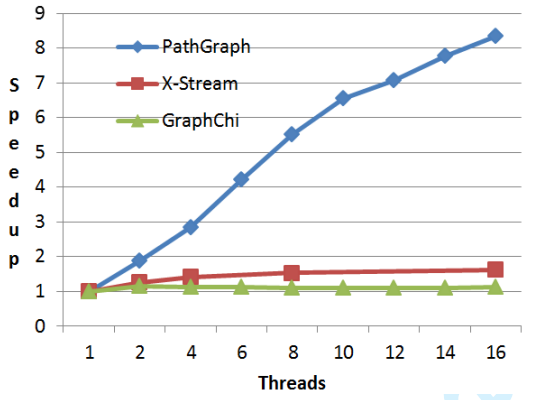


Fig. 6. PageRank Speedup

9, we reported the minimal, maximal, median and standard deviation (σ) of the relative value of the execution time of 16 threads to a random chosen thread's execution time. In Cilk, there is usually 16% - 34% difference between a long and a short execution time. For our approach, the variance between the long execution and short execution is only 0.4%-6%. The standard deviation in the execution time of 16 threads explains the significant difference in load balance between Cilk Plus and our implementation of *meques*: the standard deviation of Cilk is 3.3-98.7 times of our approach's standard deviation. From these results, we conclude that the *meque* approach to work stealing outperforms state of the art technology.

We show speedups of PathGrath, GraphChi and X-Stream from PageRank as one representative application. Since X-Stream requires powers of 2 threads, we only run X-Stream using 1, 2, 4, 8, 16 threads. As shown in Fig. 6, the speedup of PathGraph increases to 8.3 (16 threads) with the growth of threads even though its speedup slows down after 10 threads as our execution becomes memory-bound. However, the speedups of GraphChi and X-Stream are less than 1.6 and changes very slowly while the number of threads increases to 16. The reason is that these algorithms scale poorly with increasing parallelism because computing a vertex update is cheaper than retrieving the required data from neighboring vertices [25].

9 Related Work

A fair amount of work has been engaged in graph data processing [8, 9, 12, 16, 20, 21, 23, 28, 29]. Since 2004, more than 80 systems have been proposed from both academia and the industry [7]. 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, each vertex is processed in parallel. Many abstractions based on vertex centric model have been proposed [12, 16]. Since vertex centric access is random, GraphChi [12] 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 [25]. Tian etc [24] 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 [21] 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 is adjacency matrix and the Compressed Sparse Row (CSR) format [12, 19], which is equivalent to storing the graph as adjacency sets. The research on graph databases proposed some more complicate storage structures. They usually rely on physical structures and indices to speed up the execution of graph traversals and retrievals [18, 26]. They do not, however, provide powerful computational capabilities for iterative computation [12].

A required step before processing graph is graph partitioning. Graph partitioning problem has received lots of attentions over the past decade in high performance computing [2, 11, 14, 22]. Common approaches to graph partitioning involve identifying edge-cuts or vertex-cuts. While edge-cuts result in partitions that are vertex disjoint, in vertex-cuts the vertices will be replicated on all relevant partitions. The most commonly used strategies in large-scale graph processing systems are vertex-centric hash partitioning. However, this approach has extremely poor locality [9], and incurs large amount communication across partitions. A variety of heuristic algorithms, such as the multilevel partitioning paradigms have been developed that offer different cost-quality trade-offs. One widely used example of such an approach is METIS [17].

10 Conclusions and Future Work

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. Our compact path-centric storage allows for fast loading of in-edges or out-edges of a vertex for parallel scattering or gathering vertex values. In computation tier, the path based parallel graph computation is used for promoting locality-optimized processing of very large graphs. We parallel

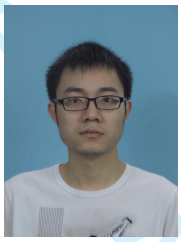
the iterative computation at partition tree level or chunk level. To provide well balanced workloads among parallel threads, we introduce the concept of multiple stealing points based task queue to allow work stealings from multiple points in the task queue. We have demonstrated that the PathGraph outperforms X-Stream and GraphChi on real graphs of varying sizes for a variety of iterative graph algorithms.

Our work on PathGraph development continues along two dimensions. First, we are working on extending PathGraph for scaling big graph using distributed computing architecture, including distributed computing algorithms and efficient communication protocols. Second, we are exploring the potential of using PathGraph to speedup the graph applications, such as natural language processing.

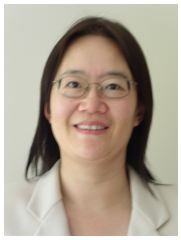
REFERENCES

[1] Cachegrind. http://www.valgrind.org/, 2014.
[2] 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.
[3] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proc. of PPoPP'13*. ACM, 2013.
[4] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proc. of SPAA'05*, pages 21–28. ACM, 2005.
[5] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proc. of SC'09*. ACM.
[6] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. BWS: Balanced work stealing for time-sharing multicores. In *Proc. of EuroSys'12*, pages 365–378. ACM, 2012.
[7] N. Doekemeijer and A. L. Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, 2014.
[8] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader. Graphct: Multithreaded algorithms for massive graph analysis. *IEEE Transactions on Parallel & Distributed Systems*, 24(11):2220–2229, 2013.
[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of OSDI'12*, pages 17–30. USENIX.
[10] Q. Gu, S. Xiong, and D. Chen. Correlations between characteristics of maximum influence and degree distributions in software networks. *Science China (Information Sciences)*, 57, 2014.
[11] G. Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *Proc. of SC'03*, pages 1–11. ACM, 2003.
[12] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proc. of OSDI'12*, pages 31–46. USENIX.
[13] Laboratory for Web Algorithmics (LAW). Datasets. http://law.di.unimi.it/datasets.php, 2013.
[14] K. Lee and L. Liu. Efficient data partitioning model for heterogeneous graphs in the cloud. In *Proc. of SC'13*, pages 25–36. ACM, 2013.
[15] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. of SPAA'10*, pages 303–314. ACM, 2010.
[16] 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.
[17] METIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
[18] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
[19] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proc. of SC'10*, pages 1–11. IEEE Computer Society, 2010.
[20] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proc. of USENIX ATC'12*, pages 4–4. USENIX, 2012.
[21] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proc. of SOSP'13*, pages 472–488. ACM, 2013.
[22] 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.
[23] J. Shun and G. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proc. of PPoPP 2013*.
[24] 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).
[25] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14):2014–2025, 2013.
[26] 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).
[27] P. Yuan, W. Zhang, C. Xie, L. Liu, H. Jin, and K. Lee. Fast iterative graph computation: A path centric approach. In *Proc. of SC 2014*, pages 401–412. IEEE, 2014.
[28] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel & Distributed Systems*, 25(8):2091–2100, 2014.
[29] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel & Distributed Systems*, 25(6):1543–1552, 2014.

Pingpeng Yuan is an associate professor in the School of Computer Science and Technology at HUST. His research interests include Semantic Web, Database etc.



Changfeng Xie is Master candidate of School of Computer Science and Technology, HUST. His current research interests include semantic web technology, massive data processing and distributed processing.



Ling Liu is a Professor in the School of Computer Science at Georgia Institute of Technology. She directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of large scale data intensive systems.

Hai Jin is a Professor and serves as Dean of the School of Computer Science and Technology at HUST. His research interests include computer architecture, virtualization technology and P2P computing.