High Performance Graph Processing with Locality Oriented Design

Pin Gao, Mingxing Zhang, Kang Chen, Yongwei Wu, *Senior Member, IEEE,* and Weimin Zheng, *Member, IEEE*

Abstract—Many distributed graph processing frameworks have emerged for helping doing large scale data analysis for many applications including social network and data mining. The existing frameworks usually focus on the system scalability without consideration of local computing performance. We have observed two locality issues which greatly influence the local computing performance in existing systems. One is the locality of the data associated with each vertex/edge. The data are often considered as a logical undividable unit and put into continuous memory. However, it is quite common that for some computing steps, only some portions of data (called as some properties) are needed. The current data layout incurs large amount of interleaved memory access. The other issue is their execution engine applies computation at a granularity of vertex. Making optimization for the locality of source vertex of each edge will often hurt the locality of target vertex or vice versa. We have built a distributed graph processing framework called Photon to address the above issues. Photon employs Property View to store the same type of property for all vertices and edges together. This will improve the locality while doing computation with a portion of properties. Photon also employs an edge-centric execution engine with Hilbert-Order that improve the locality during computation. We have evaluated Photon with 5 graph applications using 5 real-world graphs and compared it with 4 existing systems. The results show that Property View and edge-centric execution design improve graph processing by 2.4X.

Index Terms—Graph Processing, Distributed System, Property Graph.

1 Introduction

Graph processing has been increasingly popular in many areas, such as graph structure analytics, social network analytics, and recommendation systems. There have emerged many distributed graph processing frameworks, such as Pregel [1], PowerGraph [2], GraphX [3], PowerLyra [4]. They can handle graph data with extremely large size and scale well on commercial clusters.

Although scalability is important for distributed graph frameworks, the existing systems failed to provide a high local computing performance. If the performance of each server can be deeply exploited, we can achieve the same performance with less servers. With the development of network technologies, the network bandwidth has already reached Gigabytes (10Gbps) per second [5]. In high performance network, the ratio of time spent on computation is larger than that of relative lower performance network like Gbps Ethernet. Thus, the performance of single server will greatly influence the overall performance. The locality issues are critical factors affecting the single server performance.

Better data locality is beneficial for improving cache efficiency. As to graph, there are two kinds of data: data associated with vertex/edge(property) and the graph structure. So the locality of graph processing is related to the locality of properties and the locality of graph structure.

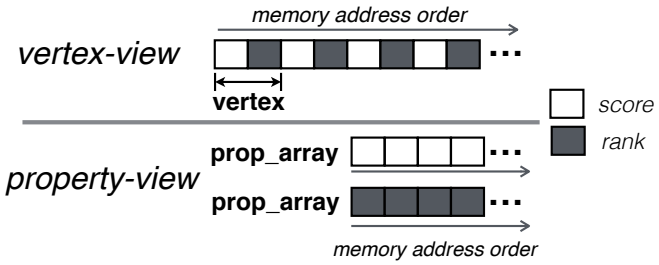


Fig. 1. Vertex View and Property View of Data Organization

There are may graph algorithms that have many properties defined on vertex or edge, such as TrustRank [6], SVDPP [7], Topic Sensitive PageRank [8], GeMV [9], AdPredictor [10], etc. Property locality refers to the locality of static data organization. It is apparent that we should store data continuously if they are accessed continuously. To this end, existing systems always store the properties of a(n) vertex/edge continuously, as they assumed that all the properties of a(n) vertex/edge will be accessed as an indivisible unit. We denote it as **Vertex View** property organization. However, this assumption is not true for many real-world cases. Taking TrustRank as an example. There are two properties defined for each vertex, one is the *rank* property that denotes the final rank of this vertex (webpage), and the other is the *score* property which represents the importance of this vertex. During one computation phase of TrustRank, both of these properties are accessed. However, for the other phase, only the *rank* property is accessed. As the result, Vertex View would introduce interleaved memory access. It will result

- *Pin Gao, Mingxing Zhang, Kang Chen, Yongwei Wu, Weimin Zheng are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China, Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China. E-mail: {gao-p12@mails., zhangmx12@mails., chenkang@, wuyw@, zwmdcs@}tsinghua.edu.cn.*
- *Corresponding author: Kang Chen and Yongwei Wu.*

in low cache efficiency due to the increased TLB misses and unnecessary data cache loads.

We propose **Property View** to organize vertex data. It stores each property separately. As Figure 1 shows, in traditional frameworks, *score* and *rank* properties, are stored together for each vertex. In contract, with Property View, they are stored separately. Property *score* of all vertices are packed together and so do all property *rank*. Thus, interleaved memory access can be avoided when only *rank* or only *score* is needed during computation.

As for graph locality, we mean the graph traverse order. To apply user defined function to the whole graph, a graph processing framework needs to traverse over all the vertices and edges. If access sequence is ordered properly, cache efficiency can be improved due to the reuse of vertex's data that has already loaded into cache. For a graph framework, execution engine controls the graph traverse order. Current distributed graph frameworks usually adopt the vertex-centric execution engine, where computation are applied at vertex granularity. It usually has better locality for vertices on one side of an edge and poor locality for vertices on the other side.

We propose an **Edge-centric Execution Engine** to improve the graph locality. The computation is applied at an edge granularity. Execution engine iterate over edges and access vertex data on both sides of every edge. Properly ordering the edges can achieve better locality for all vertices. Applying Hilbert Order to all edges is good for locality, but with the cost of sacrificing overlapping computation and communication. Photon's engine employed a **striping strategy** to achieve both better locality and overlapping.

Based on the above techniques, in this paper, we first make a comprehensive analysis of the locality and cache efficiency of Vertex View and traditional vertex-centric execution engine. Then, we present Photon, a new distributed graph processing framework that exploits cache to improve performance. It employs both the Property View design and edge-centric execution engine. The evaluations have shown that Photon outperforms state-of-the-art graph processing system 4X on average on real-world graphs for graph analytics and MLDM (Machine Learning and Data Mining) applications.

In summary, Photon aims to improve the locality of graph processing. It makes the following contributions:

1) We propose Property View data organization to maximize property locality of graph processing. It stores each property of vertex/edge data separately. With this design, interleaved memory accessed is avoided.

2) We propose edge-centric execution engine to improve graph locality. It also overlaps computation and communication in a distributed environment. This edge-centric design enabled edge ordering. Hilbert Order is adopted to achieve better locality for both source and target vertices of an edge.

3) We have done comprehensive evaluations to demonstrate the performance improvements of Photon.

2 Background and Motivation

In this section, we first explain Vertex View property organization and vertex-centrix execution along with their locality issues. Then, we illustrate why prior distributed graph frameworks fall short.

```
Gather(D_v, D_u):
  return D_u.rank / D_u.degree
Acc(a,b) :
  return a+b;
Apply(D_v, acc) :
  D_v.rank = 0.85 * acc + 0.15 * D_v.score
Scatter(D_v, D_u):
  if(!converged(D_v))
    activate(u)
```

Fig. 2. Sample Code of TrustRank on PowerGraph

2.1 Property Graph and Vertex Data Organization

In almost all the graph processing systems, the graph data is modeled and represented by **property graphs**, where there can be arbitrary number of user-defined properties associated with each vertex or edge.

Vertex View is widely adopted in vertex-centric graph frameworks such as Pregel, GraphLab, PowerGraph, and PowerLyra. In these systems, all the property data are stored in a collection of vertex data and each vertex data contains many kinds of property data. Vertex View is a natural way to express property data. However, for applications that only access part of these properties, Vertex View may lead to unsatisfactory cache performance.

Here we take TrustRank as an illustration of the disadvantages of Vertex View. TrustRank is a variation of PageRank and is widely used for identifying spam web pages. It recursively defines the rank of a webpage as Equation 1 shows. There are two properties for TrustRank, *rank* and *score*. R_i and $Score_i$ denote the rank and static score of vertex i, respectively. $Score_i$ will not change between iterations. α is a constant value named *decay factor*. $Degree_i$ denotes the out degree of vertex i.

$$R_j = (1-\alpha)Score_j + \alpha \sum_{(i,j)\in G} R_i \,/\, Degree_i \qquad (1)$$

GAS [11] is one of the programming models that takes Vertex View data organization. It has three computation stages, namely "Gather", "Apply" and "Scatter". Figure 2 shows TrustRank algorithm expressed by GAS. During Gather phase, each vertex reads data on neighbor vertices and edges with user defined *Gather* function. Gathered data can be accumulated by user-defined accumulate function *Acc*. In Apply phase, each vertex updates its states by executing function *Apply* with the data collected in Gather phase. At last, each vertex can send message or activate neighbor vertices in Scatter phase. As we can see, for TrustRank, only the *rank* property is needed during Gather phase, and at the Apply phase, all properties will be accessed. Interleaved memory access will be introduced by Vertex View data organization in Gather phase. However, the underlying CPU architecture is best suitable for accessing memory sequentially and successively.

We also tested the impact of interleaved memory access. The results show that the bandwidth of it is about 2X to 4X lesser than that of successive memory access. This is because memory are accessed in a cache-line granularity. If data size can not fill the whole cache line, interleaved access would generate a lot of extra cache line reads and TLB misses.

```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution

```
for (u,v) in all edges :
  compute with vertex[u], vertex[v], edge[u][v]
```

(b) Edge Centric Execution

Fig. 3. Vertex-centric vs Edge-centric Execution

Vertex View data organization will result in a nonoptimal property locality, since it can lead to interleaved memory access.

2.2 Vertex-Centric Execution

In graph processing, each vertex needs to access its neighbor vertices' data to update its own state. Current distributed graph frameworks usually adopt a vertex-centric execution as Figure 3a shows. Computation is applied at vertex granularity. During the processing, it first iterate over all the vertices, then for each of them, iterate over all of its neighbor vertices. Vertices in the first loop are usually sorted, i.e. they are accessed sequentially. However, the vertices in the second loop are accessed in a random way.

As we can see, vertex-centric execution achieves best locality for vertices in one side of an edge(i.e. vertices in first loop) and poor for the other(i.e. vertices in second loop).

2.3 Existing Systems and Issues

Many existing graph processing systems provide expressive programming interface and are able to deal with very large graphs. However, these systems have not addressed the importance of cache efficiency. Cache efficiency is greatly affected by locality. In the case of property locality, they usually take Vertex View to organize graph data. In the case of graph locality, they adopt vertex-centric execution engine. In this section, we first give a brief introduce of existing distributed graph processing systems. Then illustrate the issues with them.

Pregel [1] and its various open source relatives [12], [13], [14], employs "think like a vertex" philosophy and follows Bulk Synchronous Parallel (BSP) computation model. Computation is encoded in the form of user-defined update functions. Vertices communicate directly with one another by sending messages. Combiner can be used to reduce the message size sent to the same vertex.

GraphLab & PowerGraph also takes "think like a vertex" philosophy. GraphLab encodes computation into update functions, and for PowerGraph, the update function is decoupled into Gather, Apply and Scatter functions. Users can associate arbitrary data with each vertex or edge in the graph. Those data are organized with Vertex View. Different from Pregel, Graphlab and PowerGraph leverage distributed object as the underlying layer that will replicate vertex data (the replicated vertices are called mirror vertices) across machines. For Graphlab, communication between vertices is implemented by reading mirror vertices and syncing master vertices to mirrors.

PowerLyra is a graph processing engine based on PowerGraph. It is specially designed for natural graphs with skewed distribution. To reduce communication cost, it proposes a balanced *p-way hybrid-cut* algorithm that apply edge-cut to low-degree vertices and vertex-cut to high-degree vertices. The result shows that it can reduce the network data transfer considerably.

GraphX is a graph processing framework built on top of Apache Spark. It translates graph operators into dataflow operators such as Join, Map, and Group-by. GraphX adopts the vertex replication to reduce network data transfer and uses a vertex-cut partition to load balance between machines for skewed graphs. Graph data are wrapped by RDD (Resilient Distributed Dataset), and they naturally inherit the fault tolerance of RDD.

CombBLAS is a distributed in-memory graph processing framework. Since graph can be regarded as sparse matrix and data on vertices can be regarded as vectors/matrices, CombBLAS leverages matrix-vector/matrix-matrix operations to do graph processing. It has a better computing backend due to the linear algebra primitives.

However, Pregel's open source relatives and GraphX rely on the JVM (Java Virtual Machine) runtime, which makes them hard to give system designer the chance of CPU cache-level optimizations. In the case of vertex data organization, Pregel, GraphLab, PowerGraph, and PowerLyra, all takes the Vertex View to organize properties. GraphX and CombBLAS do not address the importance of property organization. In the case of execution, all these systems takes the vertex-centric execution.

3 Photon Programming Abstraction

Photon embraces both **Property View** and **Edge-Centric Execution Engine** to improve the locality of graph processing. In this section, we introduce Photon programming abstraction. The programming APIs include the abstraction for graph data as well as the operators. We will use a comprehensive example to show how to use the programming abstraction of Photon.

3.1 Photon Data Model

In Photon data model, graph data is represented by graph structure and data on vertices and edges. Graph structure maintains the connectivity of vertices. Each data on vertex and edge may hold many user-defined properties. With the decoupled abstraction of graph data, different type of property data can share the same graph structure. There are four data types in Photon, namely **Graph**, **Property Array**, **Property Collection**, **Triplet**.

Photon provides **Graph** (G) to represent the graph structure. G is a distributed data structure, each machine stores a subgraph structure and maintains the connection between others. Vertex/edge properties are co-located with the corresponding subgraph structure. The physical layout of data structure G will be discussed in § 4.1.

Photon introduces **Property Array** (PA) to abstract property data on vertices and edges. Property array takes Property View to organize vertex/edge data. For each property array, it holds a unique type of property for all the vertices or edges. It is a distributed array data structure that each machine stores a partition. In each partition, its elements are organized in dense array format.

For applications that deal with many properties, there may be many property arrays. For the ease of programming, Photon provides **Property Collection** (PC) which packs multiple property arrays together. It is only a wrapper of multiple property arrays and also keeps property view. Property arrays in one property collection should either be vertex properties or edge properties.

Photon provides another abstraction called **Triplet** (T). It has the form of $\langle PC_{src}, PC_{edge}, PC_{dst} \rangle$, where $PC_{src}, PC_{edge}, PC_{dst}$ denote the property collection for source vertex, edge, and target vertex, respectively. Each element of Triplet represent an edge with its source vertex and target vertex and the properties associated with them.

3.2 Photon Operators

Photon provides two key operators for graph processing, namely MAP and MRTRIPLET.

MAP(G :*graph*,
A :*PAType*,
F :(*PAType*) $\mapsto$ *void*)

For graph G, operator **MAP** applies the function F to all elements in property array A. F is the user-defined function that takes data type of elements in A as input. Since elements of one property array are partitioned over machines, each machine only applies F to its local partition of A. This operator will not introduce any network traffic.

MRTRIPLET(G :*graph*,
T :*triplet*,
F_{map} :*(TripletType)* $\mapsto$ *(SrcMsgType, DstMsgType)*,
F_{srccmb} :*(SrcMsgType, SrcMsgType)* $\mapsto$ *SrcMsgType*,
F_{dstcmb} :*(DstMsgType, DstMsgType)* $\mapsto$ *DstMsgType*,
F_{srcred} :*(SrcMsgType, SrcPCType)* $\mapsto$ *SrcPCType*,
F_{dstred} :*(DstMsgType, DstPCType)* $\mapsto$ *DstPCType*)

MRTRIPLET operator works on the given graph G and user-defined triplet T. $SrcMsgType$ and $DstMsgType$ denotes the type of message for source and target vertices. $SrcPCType$ and $DstPCType$ denote the property collection type of PC_{src} and PC_{dst} of triplet T.

There are three stages for operator MRTRIPLET, namely **map**, **combine** and **reduce**. Firstly, it applies the user-defined map function F_{map} on the given Triplet T and generates message for PC_{src} and PC_{dst} of T. Then, the messages send to the same vertices can be combined by the given combine function. F_{srccmb} and F_{dstcmb} denote the combine function for messages of $SrcMsgType$ and $DstMsgType$. At last, each property collection updates its status with these messages by the given reduce function. F_{srcred} and F_{dstred} denote the reduce function for PC_{src} and PC_{dst}. For applications that their source vertices do not need to receive message, F_{srccmb} and F_{srcred} can leave undefined. This operator works on both normal graphs and bipartite graphs.

3.3 Programming Example

The TrustRank algorithm implemented with Photon operators is as Figure 4 shows. We use the symbol "<" and ">" to mark the data members of a property collection. For TrustRank, there is no property defined on edges. The PC_{edge} of triplet T is marked as "NONE". $rank, deg, score, contrib$ are used to denotes the rank property, degree property, score property, and intermediate data.

```
// Define property array
PA_rank = new PA(double);
PA_score = new PA(double);
PA_deg = new PA(int64);
PA_ctb = new PA(double);
// Define property collection
PC_v = new PC(PA_rank, PA_deg, PA_ctb);
PC_src = new PC(PA_ctb);
PC_dst= new PC(PA_rank, PA_score);
// Define Triplet
T = new Triplet(PC_src, NONE, PC_dst);
// User defined functions.
def gen_contrib(<contrib, rank, deg>) :
  contrib = rank / deg

def map(<contrib>, <NONE>, <rank, score>):
  return <NONE>, <ctb>

def cmb(<msg1>, <msg2>) :
  msg = msg1 + msg2
  return <msg>

def red(<msg>, <rank, score>) :
  rank = 0.85 * msg + 0.15 * score
  return <rank, score>

// Main iteration
for(iter = 0; iter < niters; iter ++) {
  MAP(gen_contrib, PCv)
  MRTRIPLET(G, T, map, NONE, cmb, NONE, red)
}
```

Fig. 4. Sample Code of TrustRank with Photon Operators

In our implementation, MAP operation is used to generate intermediate data called *contrib*. Then, execute the MRTRIPLET operation on the given triplet. Since there is no message send to source vertex of an edge, the generated message for source vertex is marked with NONE. Messages to the same vertex are combined by the given cmb function. When one message is received, it is used to update the vertex's rank by the red function. The combine function F_{srccmb} and reduce function F_{srcred} are marked as "NONE", since source vertex of an edge does not receive messages.

4 The Photon Graph Processing System

Photon's design aims to improve the locality of graph processing from two aspects : property locality and graph locality. Photon employs a locality oriented distributed graph store, which maximizes the property locality. For graph locality, Photon adopts an edge-centric execution engine, where data are accessed at a fine-grained order. Photon's architecture is as Figure 5 shows.

4.1 Locality Oriented Distributed Graph Store

Photon's design of graph structure store improves locality from two aspects. On the one hand, the decoupled design of graph structure and property data may reduce interleaved data access. For MAP operator, graph structure is not needed at computation. If graph structure and property data stored as a whole and graph structure is not accessed, it will introduce a lot of interleaved memory access. On

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2652465, IEEE Transactions on Computers

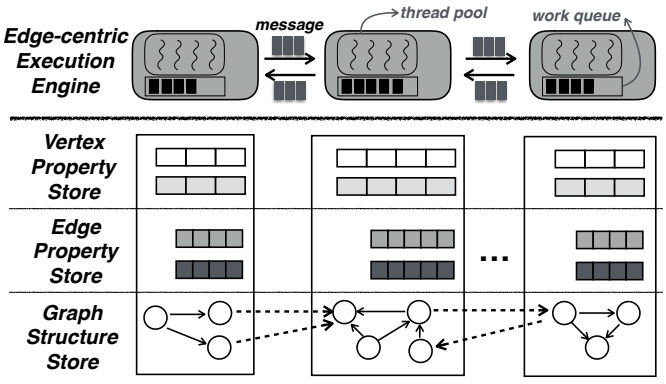
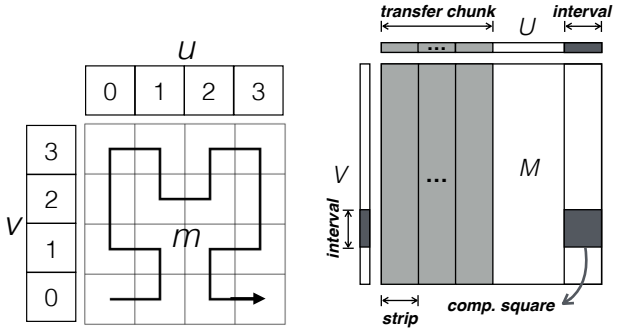


Fig. 5. Overview of Photon Architecture



(a) Hilbert Curve (b) Graph Striping

Fig. 6. Hilbert Curve and Graph Striping

the other hand, Photon uses compact data structures to represent graph structure and the mappings between subgraphs. Graph structure is stored as an array of edges in the form of $\langle VID_{src}, VID_{dst}\rangle$. VID denotes the vertex id of local graph partition, which is only valid at local domain. Since Photon adopts edge-cut [2] to partition graph into subgraphs, one vertex may have different VID at different subgraphs. A local VID to remote VID mapping is needed for data transfer. Photon represents this mapping with dense array.

Photon's property store employs Property View to organize property data defined on vertices and edges. Each kind of property data is arranged in corresponding Property Array. Data in Property Array are also stored as dense arrays, each element is with the form of $\langle D\rangle$, where D denotes the property data type. Elements in that dense array are indexed by local vertex id VID.

4.2 Edge-Centric Execution Engine

To improve graph locality, Photon has specially designed an edge-centric execution engine. It follows an edge-centric execution as Figure 3b shows. Computation is applied at edge granularity. It iterates over all the edges and then apply computation over data associated with this edge. Edges are sorted in Hilbert Order. With this design, the locality of both sides vertices of an edge can be taken into account. And as a distributed graph processing system, it overlaps computation and communication to hide the communication latency.

4.2.1 Hilbert Order and Challenges

Graph and matrix are equivalent and they can be represented by each other. Edges in a graph can be regarded as the elements of matrix in the form of $\langle row\_id, column\_id\rangle$. Source and target vertex of an edge can be considered as row and column of an elements in matrix, respectively. Thus, iterate over edges is equivalent to matrix traverse. Hilbert Order is widely used in matrix traverse because its locality-preserving property. The traverse sequence of elements in Hilbert Curve [15] is Hilbert Order. Figure 6a shows an example of Hilbert Curve within 4×4 matrix. From the view of a matrix, graph traverse with vertex ordering is equivalent to matrix traverse with row order or column order. Since Hilbert Order has proven [16] has better locality-preserving behavior than other orders, it can be used to order edges of the graph to improve the graph locality.

Unfortunately, due to one vertex's neighbor vertices are not adjacent in Hilbert Order, if it applied to the whole graph directly, computation and communication can not be overlapped. We take the Hilbert Order of Figure 6a shows as an example. v and u can be considered as source vertices and target vertices. When processed the first four edges (lower left quarter of m), data for u_0 and u_1 are updated but they can not be transmitted, because these value will also be used for the next four edges (upper left quarter of m).

4.2.2 Engine Implementation

Photon's edge-centric execution engine embraces the benefits of both Hilbert Order and overlapping by employing a **striping strategy**.

Graph Striping: Since graph is equivalent to matrix, for the ease of understanding, we illustrate striping strategy from the matrix view. Triplet in graph view can be considered as $\langle PC_{row}, PC_m, PC_{col}\rangle$ in matrix view, where PC_{row}, PC_{col} and PC_m denotes the property collection associated with row, column and matrix elements. Photon first cuts columns of M into *intervals*, as Figure 6b shows. Each interval has the same size and contains a range of continuous elements ordered by its column id. M can be cut into *stripes* by assigning elements whose column id belongs to the same interval. Each stripe has one corresponding column interval. Think about performing the MRTRIPLET operator on property collections associated with row set V, column set U and matrix M. Matrix M is cut into a set of stripes m_i ($0 \leq i < n$, where n denotes the number of stripes). U can also be cut into small units u_i which associated with the interval i. Our engine executes user-defined functions stripe by stripe. When the data of u_i is ready, the results can be transmitted immediately. Alternatively, when the data of u_i which sent from other machines arrived, the computation can get executed immediately. Through this way, the computation and communication can be overlapped.

Computation Squares. Each computation square is an $|I| \times |I|$ square matrix, where $|I|$ denotes the interval size. Thus each stripe contains many computations squares. Hilbert Order is applied on the non-zero elements in each computation square.

With the stripped design of Photon's execution engine, locality can be held inside each computation square and overlapping of computation and communication can be achieved among stripes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2652465, IEEE Transactions on Computers

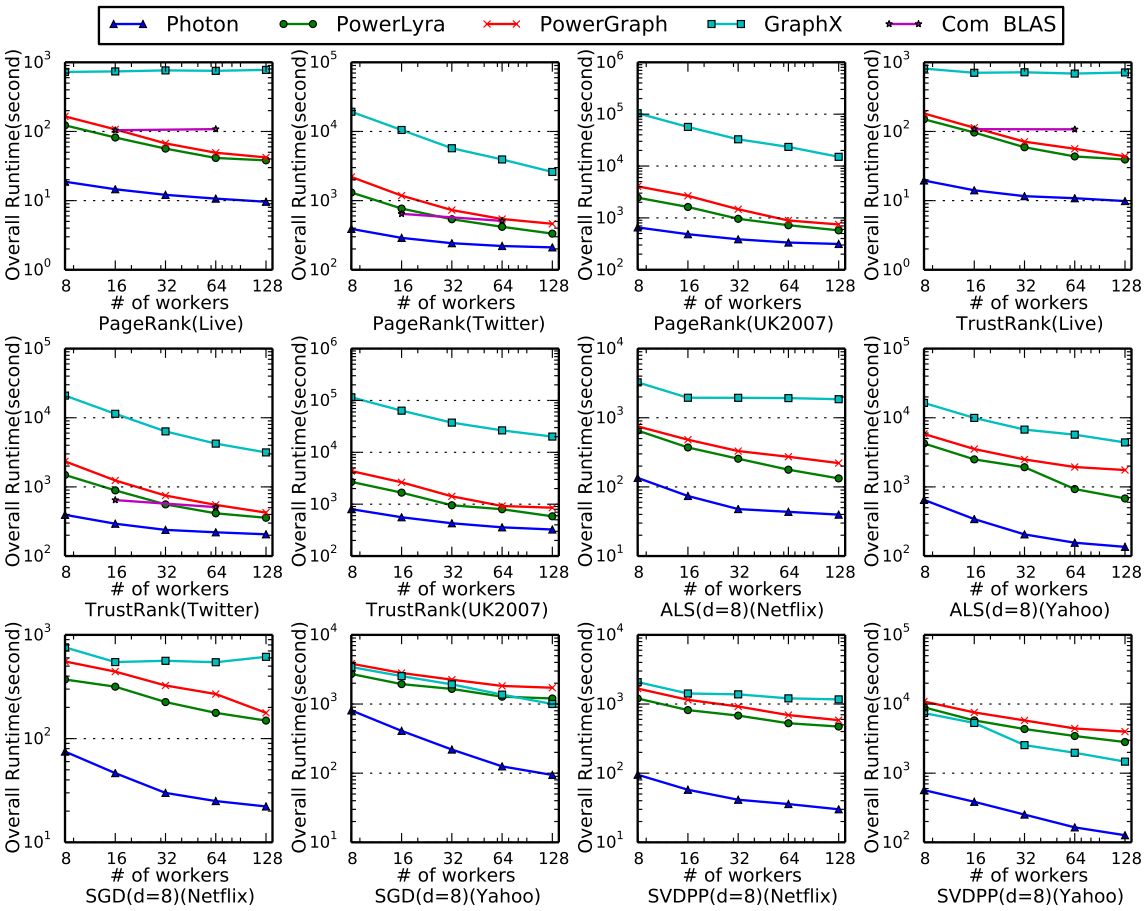


Fig. 7. Overall runtime(includes preprocessing time) comparison of Photon, PowerGraph, PowerLyra, GraphX, and CombBLAS. Each algorithm run for 50 iterations. X-axis denotes the number of workers, and y-axis is the execution time in seconds in log scale.

TABLE 1
A Collection of Real World Graphs

Graph	$\|V\|$	$\|E\|$	Pre-Time(s)	Sort-Time(s)
LiveJournal [17]	4.85M	69.0M	7.95	0.33
Twitter [18]	42M	1.5B	210.62	7.11
UK-2007 [19]	105.9M	3.7B	313.59	14.34
Netflix [20]	0.5M	100M	15.50	0.92
YahooMusic [21]	1.9M	717M	43.58	7.23

5 Evaluation

In this chapter, we present the detailed evaluation results to support our design choices. Also, we compare our system with state of the art graph processing frameworks to demonstrate the efficiency of Photon.

We first introduce the experiment setup. Then, the rest sections answer the following questions: 1) Comparing to the existing systems, how well does Photon perform? 2) Comparing to Vertex View, how much does Property View benefits the computing? 3) What about the performance of Photons edge-centric execution engine? 4) What is the impaction of overlapping communication and computation in Photon? 5) What is the scalability of Photon?

5.1 Experiment Setup

We evaluate Photon on a cluster of commodity multi-core machines. Each machine has two Intel Xeon CPU E5-2640 v2 (16 physical cores share 20MB LLC, hyper-threading is disabled), 96GB of memory. Each machine has one 1Gbps Ethernet NIC and one 40Gbps Mellanox MT27500 InfiniBand NIC for communication. The operating system is Ubuntu 14.04 with kernel version 3.13. Photon runs on top of MPICH-3.14.

We choose five graph algorithms as our testing applications, namely PageRank, TrustRank, Alternating Least Squares (ALS), Stochastic Gradient Descent (SGD), and Singular Value Decomposition Plus Plus (SVDPP). The latter three are collaborative filtering (CF) algorithms widely used in recommendation systems. For each algorithm, we run 50 iterations and report the overall runtime, including preprocessing time.

For most of the collaborative filtering algorithms, one property on the vertex data is a vector denoted as *feature vector*, the size of feature vector is denoted as *latent dimension* (d). Higher d will produce higher algorithm accuracy of prediction while on the other hand will increase the computation and communication cost.

We take 5 real-world graphs as the testing set for showing the performance results. Among these graphs, LiveJournal, Twitter, UK-2007-05 are social network graphs, which can be used to benchmark applications such as PageRank, TrustRank. The other two are user-item rating graphs (bipartite graph) and are used for evaluating the CF algorithms. Details of the graphs can be found in Table 1.

The preprocessing time is also evaluated(noted as Pre-Time), as Table 1 shows. Graph preprocessing includes graph partition, building local graphs and sorting edges

with Hilbert Order. We also evaluated preprocessing time of other frameworks. PowerGraph performs best in preprocessing, and that time for the five graphs are 8.24, 179.9, 270.9, 14.49, 30.98. We can see our preprocessing time is acceptable. Since Hilbert Order is applied to computation squares not the whole graph, the edge sorting time is not an issue. Moreover, the graph preprocessing time can be amortized over iterations of computation.

5.2 Overall Performance Comparison

We have conducted the comparison between Photon and the existing systems. The results of overall performance comparison are shown in Figure 7 [1]. The evaluation is performed on 8 machines, and we increase the number of workers (cores) on each machine.

As shown in the figure, Photon outperforms these system from 1.58X up to 10X in overall runtime. Some of these performance improvements come from the use of compact data structures in our implementation. To better illustrate the benefits of Property View and Edge-Centric Execution engine, we have done performance breakdown as following sections shows. On single server, Property View and Edge-Centric Execution engine brings 1.8X and 1.37X speedup, respectively, and they lead to a 2.4X performance speed up in total.

5.3 The Effectiveness of Property View

Here, we take TrustRank as an example. We compare the performance of the two algorithms implemented in Photon with Property View and Vertex View. In the Vertex View version, Photon manually defines those properties in one data structure, i.e. Vertex, and then treat the data structure as one property. The other version takes the property view and stores properties separately. We evaluated TrustRank on Twitter graph with multi-thread and distributed configurations. The multi-thread configuration evaluates TrustRank in a single machine with the number of threads increases. The distributed evaluation is performed on 8 machines and increases the number of threads on each machine. Results are in Figure 9. The label on each bar denotes the performance boost of Property View compared to Vertex View. From the results, we can see that Property View outperforms Vertex View up to about 1.8X.

The improvement from Property View over many properties mainly depends on the property size and the ratio of accessed properties during computing stages. For TrustRank, at the scatter computation stage, only half of properties are accessed, and this stage dominates the overall computation time. Thus, theoretically, the property view can have nearly 2X performance improvement at most. In our evaluation, single thread implementation has 1.8X performance improvement, which is acceptable. There will be more improvement while less portion of properties is accessed during computation.

1. CombBLAS only works on square number of processes. Due to the limitation of its interface, only PageRank and TrustRank can be implemented. CombBLAS failed to run on UK2007 graph because of its huge memory cost.

Fig. 8. Single Server Performance for PageRank

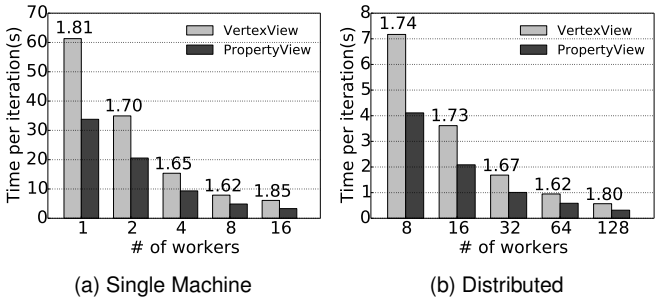


Fig. 9. Vertex View vs Property View on TrustRank

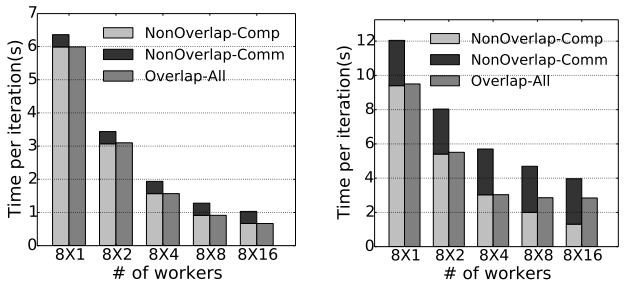
5.4 Edge-Centric Execution Engine

We evaluate Photon's edge-centric execution engine from two aspects : single server performance to show photon's edge-centric design and distributed performance to show the effect of overlapping.

We take the PageRank algorithm on LiveJournal graph to benchmark the efficiency of single server performance. We compared Photon (Pho-Hilbert) with PowerLyra, CombBLAS and edges sorted by target vertex id (Pho-Target). Figure 8 shows the results[2]. We take Pho-Hilbert as the baseline, and the labels on the bars denote the normalized execution time. From the results we can see, the edge-centric execution engine of Photon outperform PowerLyra at all configurations and outperform CombBLAS in most configurations. Furthermore, it scale well in the multi-thread environment, whereas PowerLyra and CombBLAS does not scale well with the thread number increases. With the comparison of Pho-Target and Pho-Hilbert, we can see, by taking Hilbert order, it brings about 1.3X performance improvement. From these observations, we conclude that the edge-centric design and Hilbert Order integrated with Photon's edge-centric execution engine bring notable benefits.

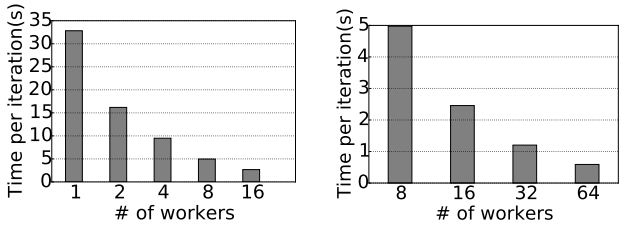
We implemented another version of Photon that do not overlap computation and communication. It records the time cost of computation and communication separately. Since different applications may have different computation-communication ratio, we benchmark two algorithms (PageRank and SVDPP with $d = 128$) to evaluate how much overlapping can benefit each algorithm. The result is in Figure 10. The x-axis shows the *number of machines* $\times$ *thread number on each machine*. We break the execution time into computation phase (gray bar) and communication phase (dark bar). We can see that with the number of threads increases, the proportion that communication takes increases. If the computation phase time is greater than the

2. CombBLAS only works with square number of processes.



(a) PageRank (b) SVDPP (d = 128)

Fig. 10. Overalap of Computation and Communication of PageRank(on UK2007 Graph) and SVDPP(on Netflix Graph)



(a) Scale-up (b) Scale-out

Fig. 11. Scale-up and Scale-out for PageRank on Twitter Graph

communication phase (for the SVDPP case), this means the communication can not be fully masked by computation. For PageRank, in our computer cluster, communication can be fully overlapped by computation.

5.5 Scalability of Photon

We evaluate the scalability of Photon from two aspects: scalability by using more threads on a single machine and scalability by adding more machines to the system. In the single machine configuration, there will be no data transfer over networking. Figure 11 shows the scalability of Photon for PageRank on Twitter graph. Photon has good scalability in both single machine configuration and distributed configuration.

6 Conclusion

This paper aims to improve the locality of graph processing from two aspects : property locality and graph locality. As for property locality, we proposed Property View property organization. It organizes each properties separately. Compared with Vertex View property organization, Property View reduces the interleaved memory access during computation. As for graph locality, we proposed edge-centric execution engine, where computation are applied at an edge granularity. It enables edge ordering and Hilbert Order can be applied to improve the graph locality. We have implemented a distributed graph processing system named Photon. Photon embraces both Property View and Edge-Centric Execution Engine to improve the cache efficiency of graph processing. Results shows that Property View brings upt o 1.8X speedup and Edge-Centric Execution Engine brings up to 1.37X speedup.

Acknowledgments

The paper benefited greatly from the insightful feedback from Jinglei Ren on an early version of the paper. This Work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, U1435216), National Basic Research (973) Program of China (2014CB340402).

References

[1] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[3] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of OSDI*, 2014, pp. 599–613.

[4] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 1.

[5] "InfiniBand," https://en.wikipedia.org/wiki/InfiniBand, [Online; accessed 27-September-2015].

[6] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen, "Combating web spam with trustrank," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 576–587.

[7] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 426–434.

[8] T. H. Haveliwala, "Topic-sensitive pagerank," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 517–526.

[9] "cblas_gemv," https://software.intel.com/en-us/node/520750, [Online; accessed 27-September-2015].

[10] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 13–20.

[11] 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," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[12] "The Apache Hama Project," http://hama.apache.org/, [Online; accessed 27-September-2015].

[13] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.

[14] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[15] D. Hilbert, "Ueber die stetige abbildung einer line auf ein flächenstück," *Mathematische Annalen*, vol. 38, no. 3, pp. 459–460, 1891.

[16] H. Haverkort and F. van Walderveen, "Locality and bounding-box quality of two-dimensional space-filling curves," *Computational Geometry*, vol. 43, no. 2, pp. 131–147, 2010.

[17] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 44–54.

[18] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

[19] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," in *ACM SIGIR Forum*, vol. 42, no. 2. ACM, 2008, pp. 33–38.

[20] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management*. Springer, 2008, pp. 337–348.

[21] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r, [Online; accessed 27-September-2015].