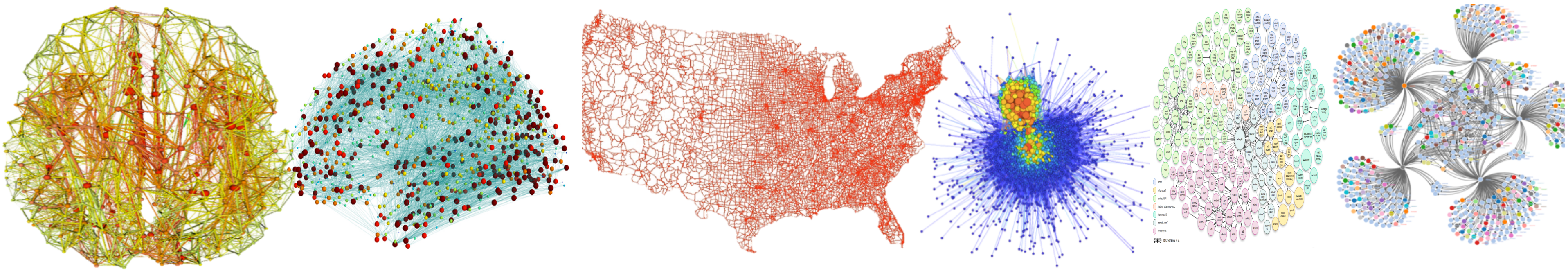


GraphLego

Fast Iterative Graph Computation with Resource Aware Graph Parallel Abstraction

Yang Zhou, Ling Liu, Kisung Lee, Calton Pu, Qi Zhang

Distributed Data Intensive Systems Lab (DiSL)

Georgia Tech College of Computing

GraphLego: Motivation

- **Graphs are everywhere**: Internet, Web, Road networks, Protein Interaction Networks, Utility Grids
- **Scale of Graphs studied in literature**: billions of edges, tens/hundreds of GBs

[Paul Burkhardt, Chris Waring 2013]

Popular graph datasets in current literature

	n (vertices in millions)	m (edges in millions)	size
AS-Skitter	1.7	11	142 MB
LJ	4.8	69	337.2 MB
USRD	24	58	586.7 MB
BTC	165	773	5.3 GB
WebUK	106	1877	8.6 GB
Twitter	42	1470	24 GB
YahooWeb	1413	6636	120 GB

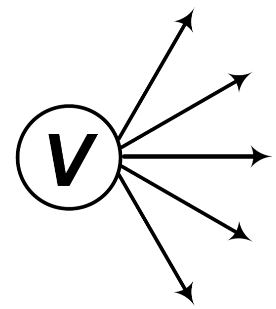
- Brain Scale: 100 billion vertices, 100 trillion edges

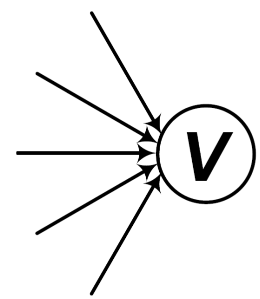
Existing Graph Processing Systems

- Single PC Systems
  - **GraphLab** [Low et al., UAI'10]
  - **GraphChi** [Kyrola et al., OSDI'12]
  - **X-Stream** [Roy et al., SOSP'13]
  - **TurboGraph** [Han et al., KDD'13]
- Distributed Shared Memory Systems
  - **Pregel** [Malewicz et al., SIGMOD'10]
  - **Giraph/Hama** – Apache Software Foundation
  - **Distributed GraphLab** [Low et al., VLDB'12]
  - **PowerGraph** [Gonzalez et al., OSDI'12]
  - **SPARK-GraphX** [Gonzalez et al., OSDI'14]

Vertex-centric Computation Model

- Think like a vertex
- vertex_scatter(vertex *v*)
  - send updates over outgoing edges of *v*
- vertex_gather(vertex *v*)
  - apply updates from inbound edges of *v*





- repeat the computation iterations
  - for all vertices *v*
    - vertex_scatter(*v*)
  - for all vertices *v*
    - vertex_gather(*v*)

Edge-centric Computation Model (X-Stream)

- Think like an edge (source vertex and destination vertex)
- edge_scatter(edge *e*)
  - send update over *e* (from source vertex to destination vertex)
- update_gather(update *u*)
  - apply update *u* to *u.destination*

- repeat the computation iterations
  - for all edges *e*
    - edge_scatter(*e*)
  - for all updates *u*
    - update_gather(*u*)

Challenges of Big Graphs

- **Graph size v.s. limited resource**
  - Handling big graphs with billions of vertices and edges in memory may require hundreds of gigabytes of DRAM
- **High-degree vertices**
  - In uk-union with 133.6M vertices: the maximum indegree is 6,366,525 and the maximum outdegree is 22,429
- **Skewed vertex degree distribution**
  - In Yahoo web with 1.4B vertices: the average vertex degree is 4.7, 49% of the vertices have degree zero and the maximum indegree is 7,637,656
- **Skewed edge weight distribution**
  - In DBLP with 0.96M vertices: among 389 coauthors of Elisa Bertino, she has only one coauthored paper with 198 coauthors, two coauthored papers with 74 coauthors, three coauthored papers with 30 coauthors, and coauthored paper larger than 4 with 87 coauthors

Real-world Big Graphs

Graph	Type	#Vertices	#Edges	AvgDeg	MaxIn	MaxOut
Yahoo	directed	1.4B	6.6B	4.7	7.6M	2.5K
uk-union	directed	133.6M	5.5B	41.22	6.4M	22.4K
uk-2007-05	directed	105.9M	3.7B	35.31	975.4K	15.4K
Twitter	directed	41.7M	1.5B	35.25	770.1K	3.0M
Facebook	undirected	5.2M	47.2M	18.04	1.1K	1.1K
DBLPS	undirected	1.3M	32.0M	40.67	1.7K	1.7K
DBLPM	undirected	0.96M	10.1M	21.12	1.0K	1.0K
Last.fm	undirected	2.5M	42.8M	34.23	33.2K	33.2K

Graph Processing Systems: Challenges

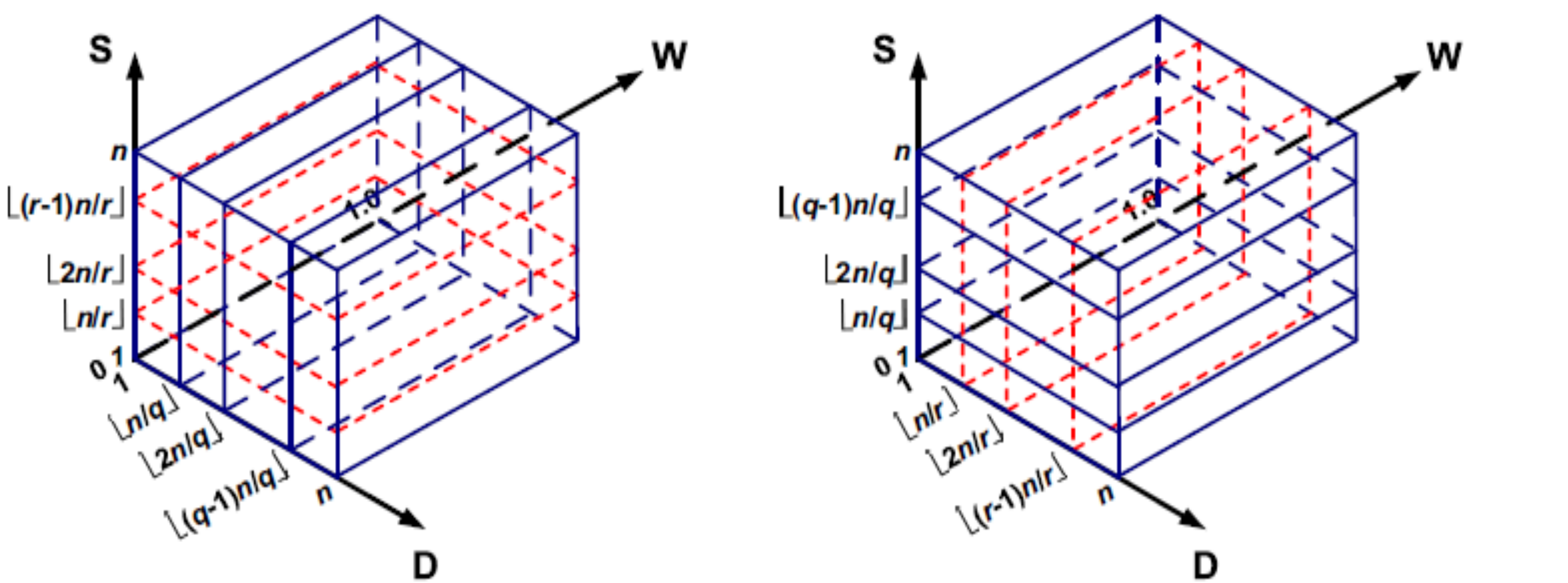
- **Diverse types of processed graphs**
  - Simple graph: not allow for parallel edges (multiple edges) between a pair of vertices
  - Multigraph: allow for parallel edges between a pair of vertices
- **Different kinds of graph applications**
  - Matrix-vector multiplication and graph traversal with the cost of $O(n^2)$
  - Matrix-matrix multiplication with the cost of $O(n^3)$
- **Random access**
  - It is inefficient for both access and storage. A bunch of random accesses are necessary but would hurt the performance of graph processing systems
- **Workload imbalance**
  - The time of computing on a vertex and its edges is much faster than the time to access to the vertex state and its edge data in memory or on disk
  - The computation workloads on different vertices are significantly imbalanced due to the highly skewed vertex degree distribution.

GraphLego: Our Approach

- **Flexible multi-level hierarchical graph parallel abstractions**
  - Model a large graph as a 3D cube with source vertex, destination vertex and edge weight as the dimensions
  - Partitioning a big graph by: slice, strip, dice based graph partitioning
- **Access Locality Optimization**
  - Dice-based data placement: store a large graph on disk by minimizing non-sequential disk access and enabling more structured in-memory access
  - Construct partition-to-chunk index and vertex-to-partition index to facilitate fast access to slices, strips and dices
  - implement partition-level in-memory gzip compression to optimize disk I/Os
- **Optimization for Partitioning Parameters**
  - Build a regression-based learning model to discover the latent relationship between the number of partitions and the runtime

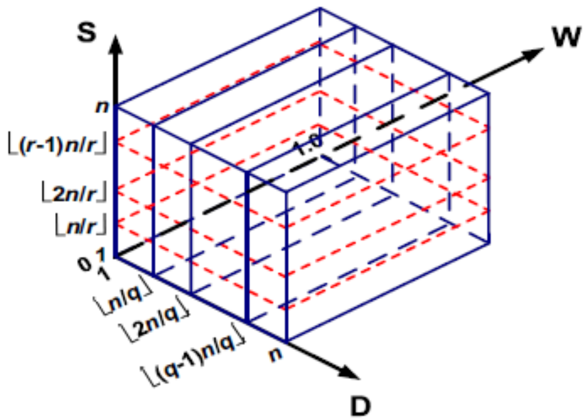
Modeling a Graph as a 3D Cube

- Model a directed graph *G*=(*V*,*E*,*W*) as a 3D cube *I*=(*S*,*D*,*E*,*W*) with source vertices (*S*=*V*), destination vertices (*D*=*V*) and edge weights (*W*) as the three dimensions

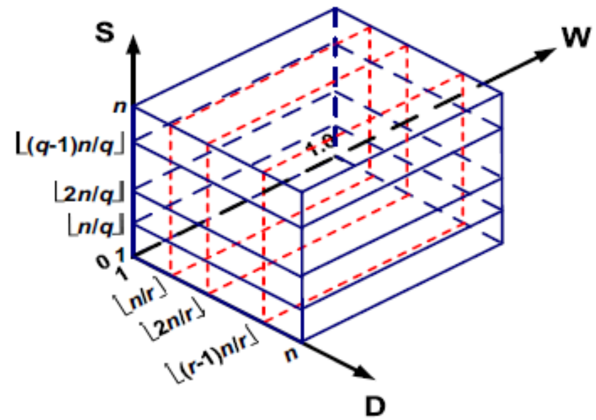


(a) In-edge Cube (b) Out-edge Cube

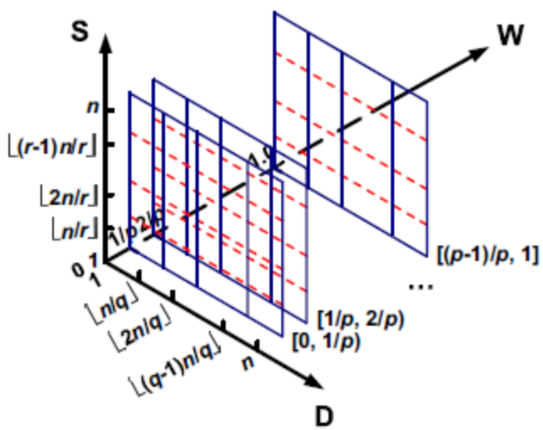
Multi-level Hierarchical Graph Parallel Abstractions



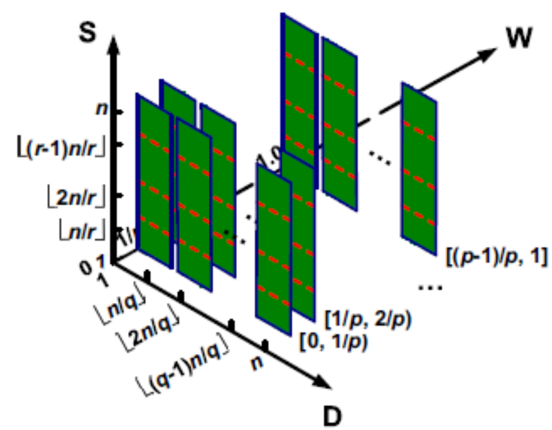
(a) In-edge Cube



(b) Out-edge Cube

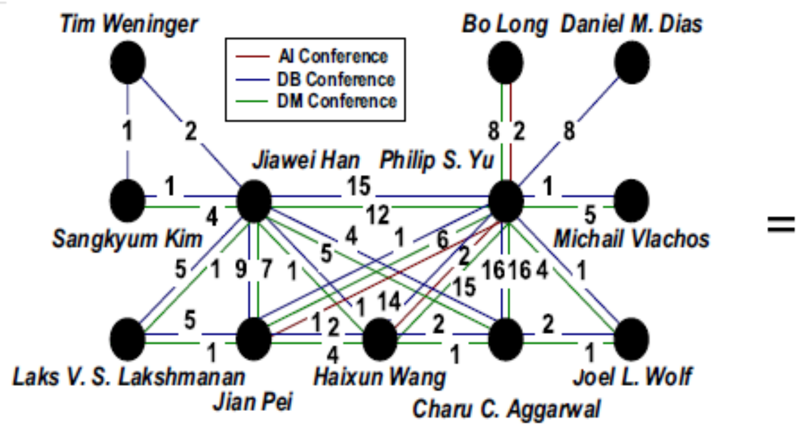


(c) In-edge Slice

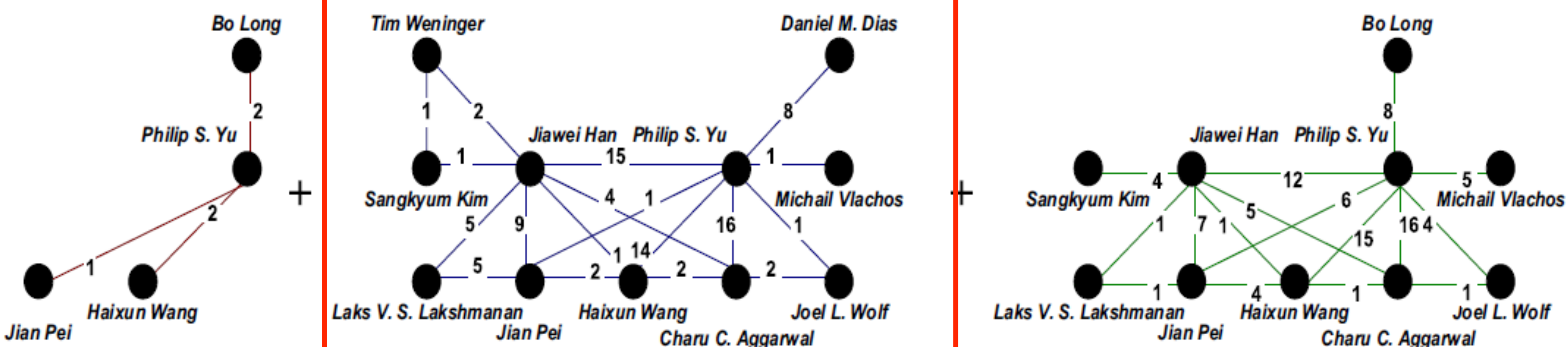


(d) In-edge Strip

Slice Partitioning: DBLP Example

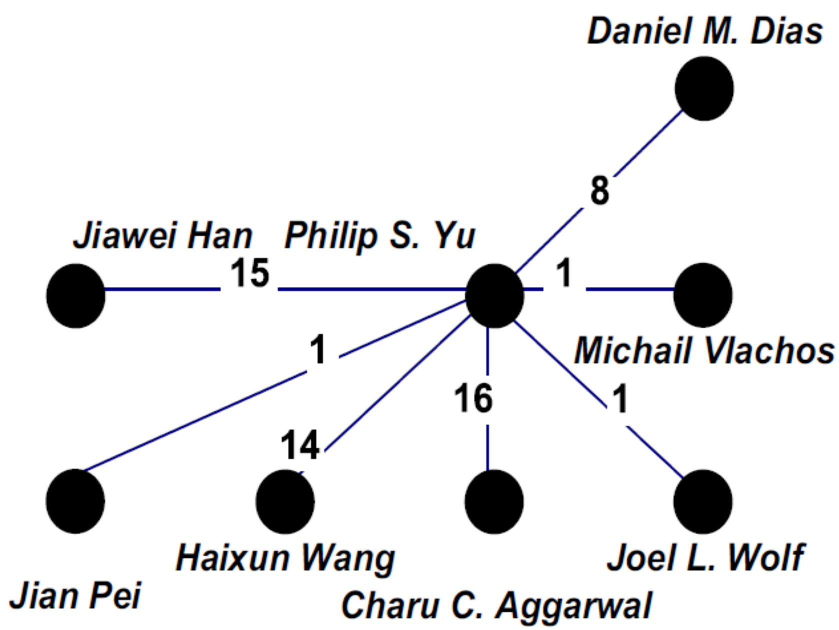


(a) Coauthor Multigraph

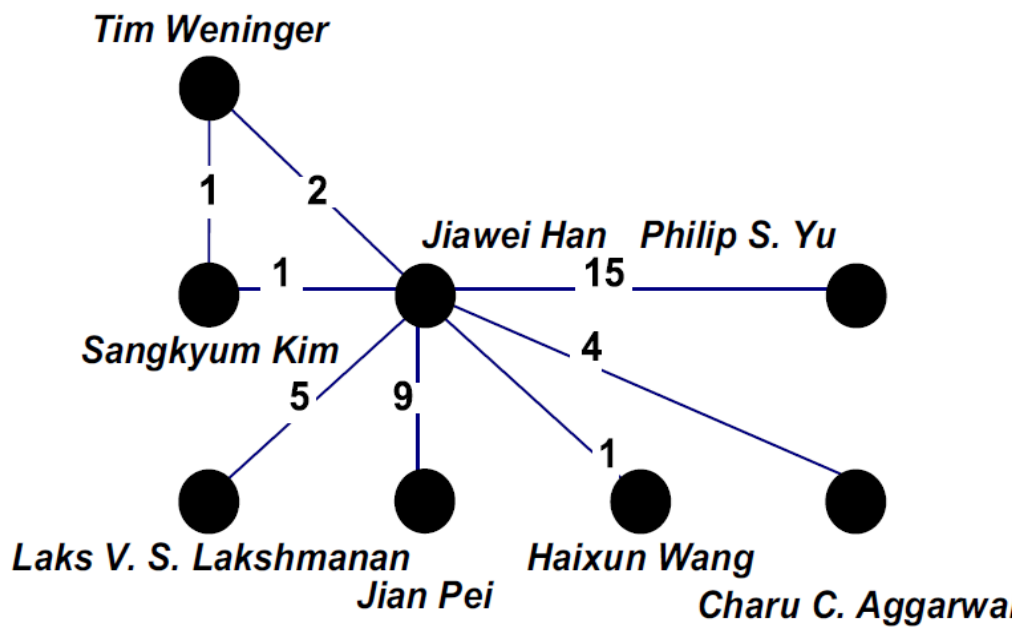


(b) AI Slice (c) DB Slice (d) DM Slice

Strip Partitioning of DB Slice

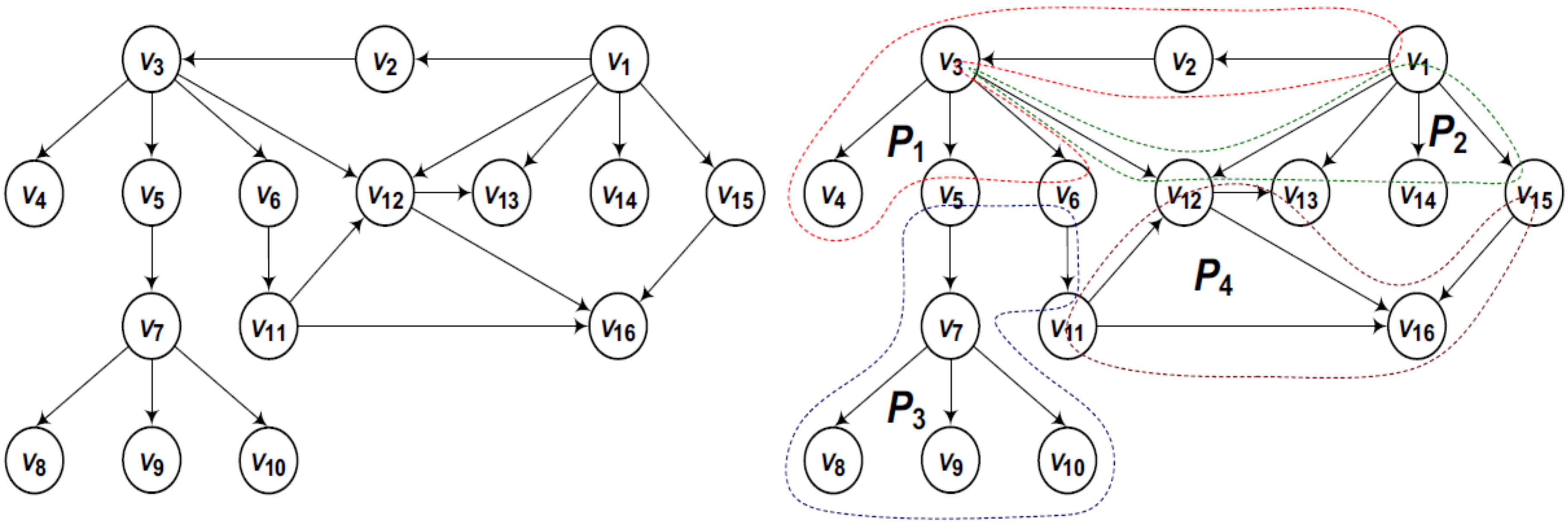


(a) Strip 1



(b) Strip 2

Dice Partitioning: An Example



(a) Original Graph (b) Dice Partitioning

SVP: v1,v2,v3, v5,v6,v7, v11, v12, v15

DVP: v2,v3,v4,v5,v6, v7,v8,v9,v10,v11, v12,v13,v14,v15.v16

Dice Partition Storage (OEDs)

Vertex Table

SVP 1: V_1 V_2 V_3
SVP 2: V_5 V_6 V_7
SVP 3: V_{11} V_{12} V_{15}

DVP 1: V_2 V_3 V_4 V_5 V_6
DVP 2: V_7 V_8 V_9 V_{10} V_{11}
DVP 3: V_{12} V_{13} V_{14} V_{15} V_{16}

P_1: *SVP* 1 $\cup$ *DVP* 1
P_2: *SVP* 1 $\cup$ *DVP* 3
P_3: *SVP* 2 $\cup$ *DVP* 2
P_4: *SVP* 3 $\cup$ *DVP* 3

Edge Table

P_1	P_2
(V_1, V_2)	(V_1, V_{12})
(V_2, V_3)	(V_1, V_{13})
(V_3, V_4)	(V_1, V_{14})
(V_3, V_5)	(V_1, V_{15})
(V_3, V_6)	(V_3, V_{12})

P_3	P_4
(V_5, V_7)	(V_{11}, V_{12})
(V_6, V_{11})	(V_{11}, V_{16})
(V_7, V_8)	(V_{12}, V_{13})
(V_7, V_9)	(V_{12}, V_{16})
(V_7, V_{10})	(V_{15}, V_{16})

Vertex Map

V_1	P_1, P_2
V_2	P_1
V_3	P_1, P_2
V_4	P_1
V_5	P_1, P_3
V_6	P_1, P_3
V_7	P_3
V_8	P_3
V_9	P_3
V_{10}	P_3
V_{11}	P_3, P_4
V_{12}	P_2, P_4
V_{13}	P_2, P_4
V_{14}	P_2
V_{15}	P_2, P_4
V_{16}	P_4

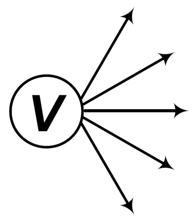
Advantage: Multi-level Hierarchical Graph Parallel Abstractions

- Choose smaller subgraph blocks such as dice partition or strip partition to balance the parallel computation efficiency among partition blocks
- Use larger subgraph blocks such as slice partition or strip partition to maximize sequential access and minimize random access

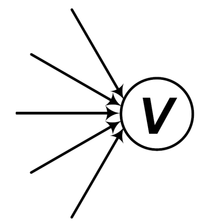
Programmable Interface

- vertex centric programming API, such as *Scatter* and *Gather*.

Scatter vertex updates to outgoing edges



Gather vertex updates from neighbor vertices and incoming edges



- Compile iterative algorithms into a sequence of internal function (routine) calls that understand the internal data structures for accessing the graph by different types of subgraph partition blocks

Algorithm 2 PageRank

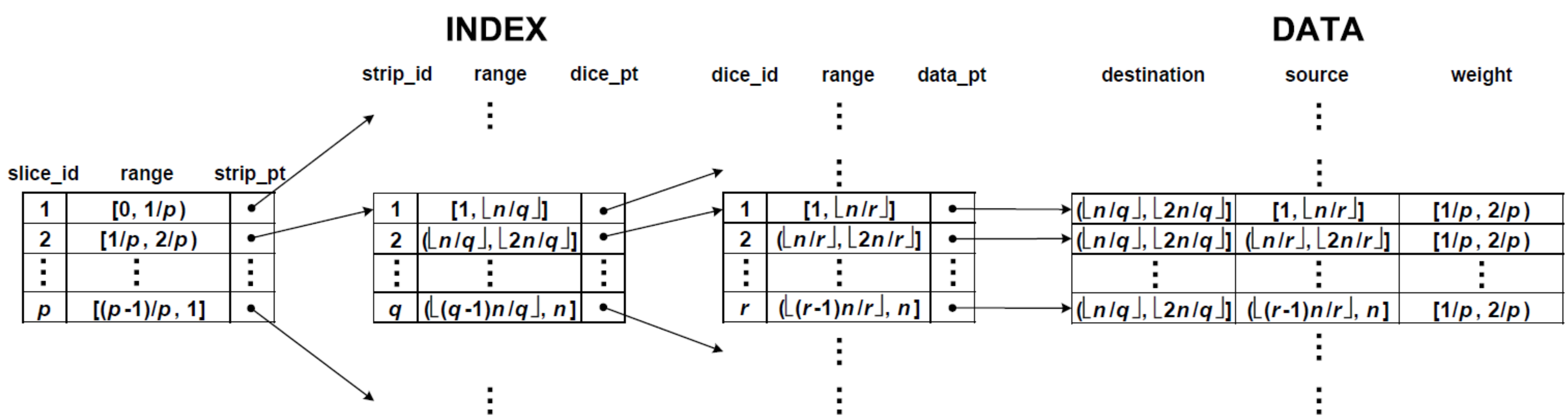
```
Initialize(v)
    v.rank = 1.0;

Scatter(v)
    msg = v.rank/v.degree;
    //send msg to destination vertices of v's out-edges

Gather(v)
    state = 0;
    for each msg of v
    //receive msg from source vertices of v's in-edges
      state += msg; //summarize partial vertex updates
    v.rank = 0.15+0.85*state; //produce complete vertex update
```

Partition-to-chunk Index
Vertex-to-partition Index

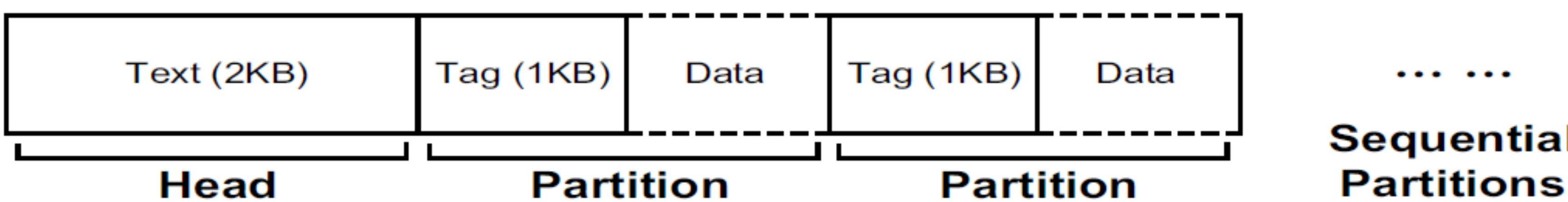
- The dice-level index is a dense index that maps a dice ID and its DVP (or SVP) to the chunks on disk where the corresponding dice partition is stored physically
- The strip-level index is a two level sparse index, which maps a strip ID to the dice-level index-blocks and then map each dice ID to the dice partition chunks in the physical storage
- The slice level index is a three-level sparse index with slice index blocks at the top, strip index blocks at the middle and dice index blocks at the bottom, enabling fast retrieval of dices with a slice-specific condition



Partition-level Compression

- Iterative computations on large graphs incur non-trivial cost for the I/O processing
  - The I/O processing of Twitter dataset on a PC with 4 CPU cores and 16GB memory takes 50.2% of the total running time for PageRank (5 iterations)

- Apply in-memory gzip compression to transform each graph partition block into a compressed format before storing them on disk



Data Structure for Compressed Graph

Configuration of Partitioning Parameters

- **User definition**
- **Simple estimation**
- **Regression-based learning**
  - Construct a polynomial regression model to model the nonlinear relationship between independent variables p, q, r (partition parameters) and dependent variable T (runtime) with latent coefficient α_{ijk} and error term ε

$$T \approx f(p,q,r,\alpha) = \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \sum_{k=1}^{n_r} \alpha_{ijk} p^i q^j r^k + \epsilon$$

  - The goal of regression-based learning is to determine the latent α_{ijk} and ε to get the function between p, q, r and T
  - Select m limited samples of (p_l, q_l, r_l, T_l) $(1 \le l \le m)$ from the existing experiment results
  - Solve m linear equations consisting of m selected samples to generate the concrete α_{ijk} and ε
  - Utilize a successive convex approximation method (SCA) to find the optimal solution (i.e., the minimum runtime T) of the above polynomial function and the optimal parameters (i.e., p, q and r) when T is minimum

$$T_1 = \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \sum_{k=1}^{n_r} \alpha_{ijk} p_1^i q_1^j r_1^k + \epsilon$$

$$\cdots \qquad \cdots$$

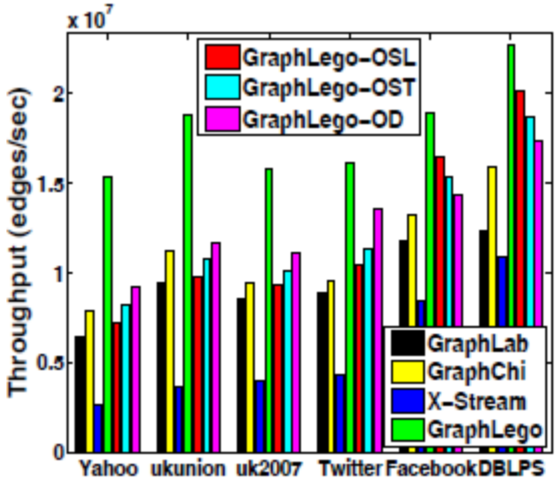
$$T_m = \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \sum_{k=1}^{n_r} \alpha_{ijk} p_m^i q_m^j r_m^k + \epsilon$$

Experimental Evaluation

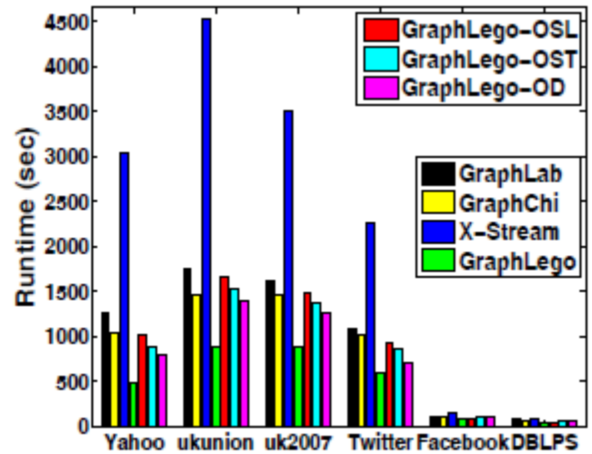
- Computer server
  - Intel Core i5 2.66 GHz, 16 GB RAM, 1 TB hard drive, Linux 64-bit
- Graph parallel systems
  - **GraphLab** [Low et al., UAI'10]
  - **GraphChi** [Kyrola et al., OSDI'12]
  - **X-Stream** [Roy et al., SOSP'13]
- Graph applications

Application	Propagation	Core Computation
PageRank	single graph	matrix-vector
SpMV	single graph	matrix-vector
Connected Components	single graph	graph traversal
Diffusion Kernel	two graphs	matrix-matrix
Inc-Cluster	two graphs	matrix-matrix
Matrix Multiplication	two graphs	matrix-matrix
LMF	multigraph	matrix-vector
AEClass	multigraph	matrix-vector

Execution Efficiency on Single Graph

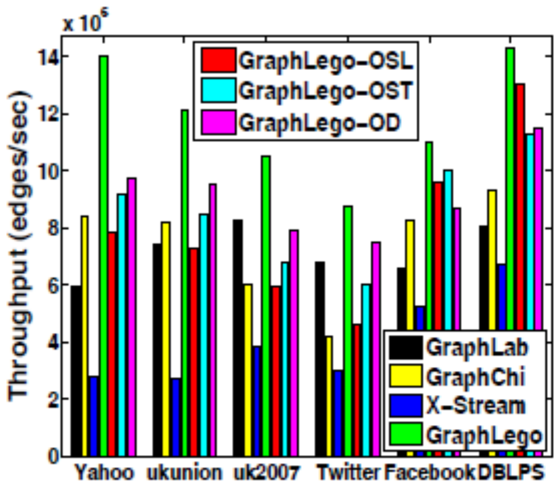


(a) Throughput

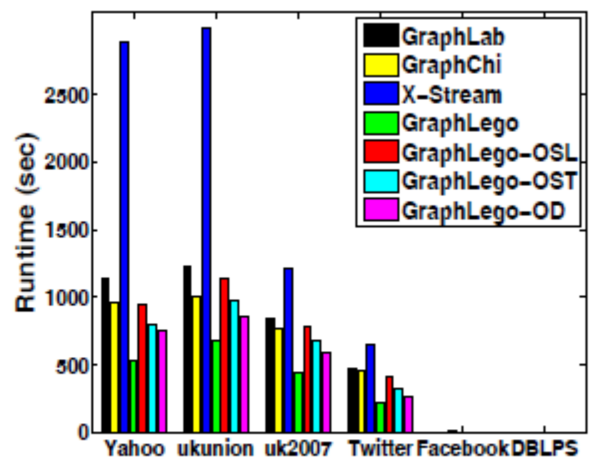


(b) Runtime

PageRank on Six Real Graphs



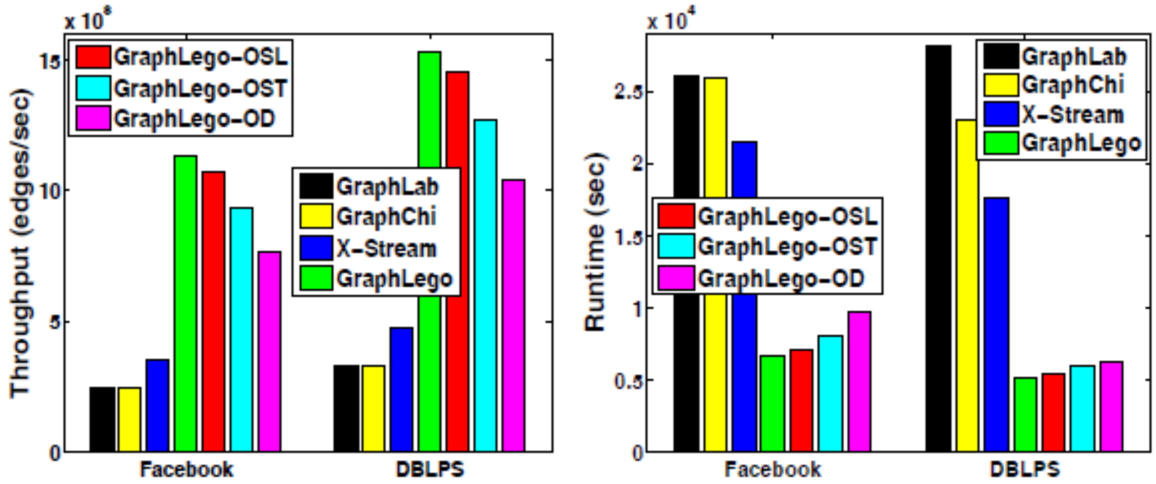
(a) Throughput



(b) Runtime

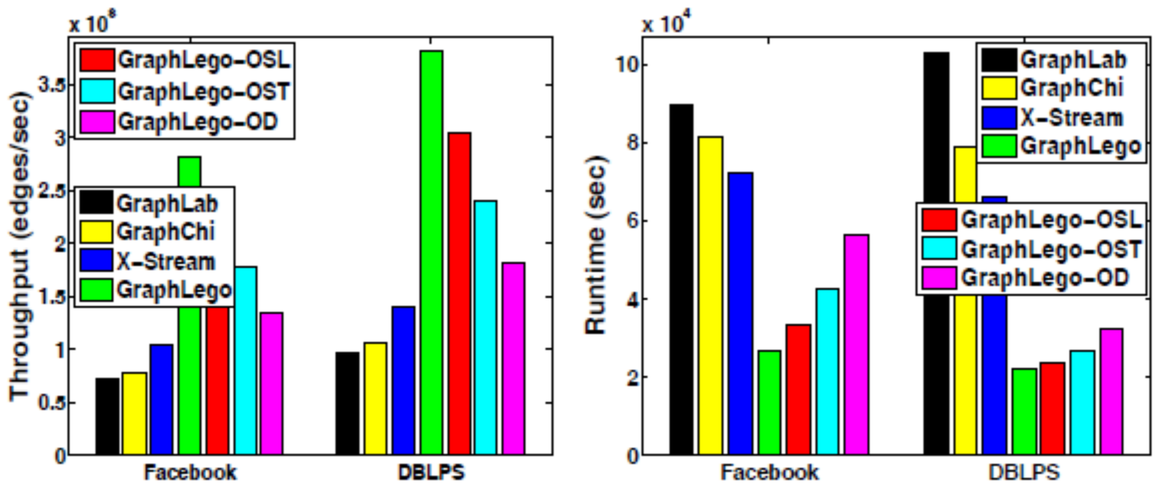
SpMV on Six Real Graphs

Execution Efficiency on Multiple Graphs



(a) Throughput (b) Runtime

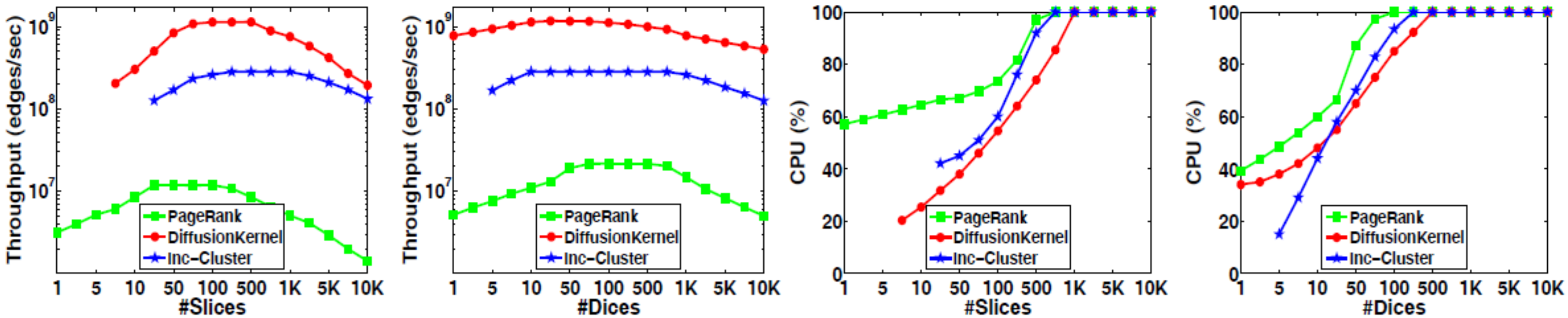
Diffusion Kernel on Two Real Graphs



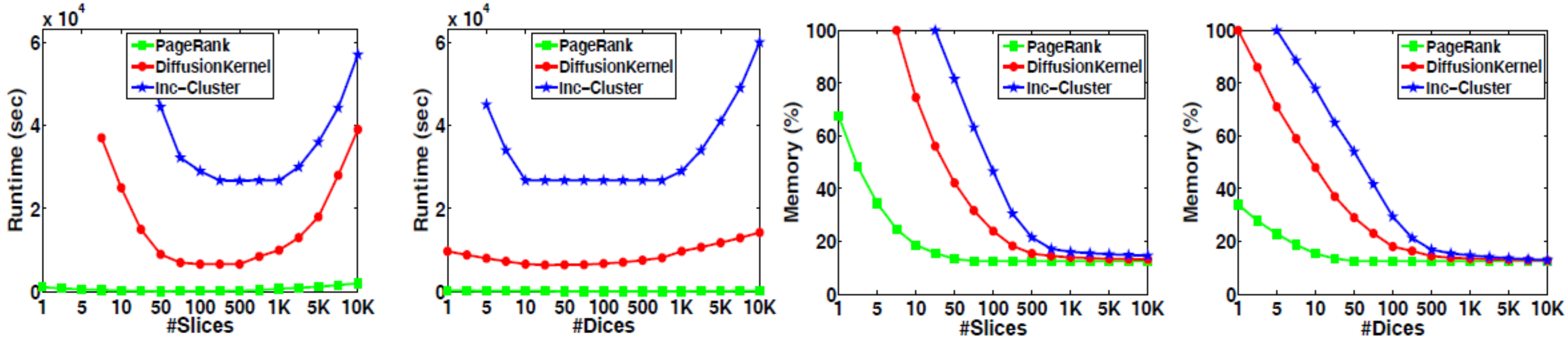
(a) Throughput (b) Runtime

Inc-Cluster on Two Real Graphs

Decision of #Partitions

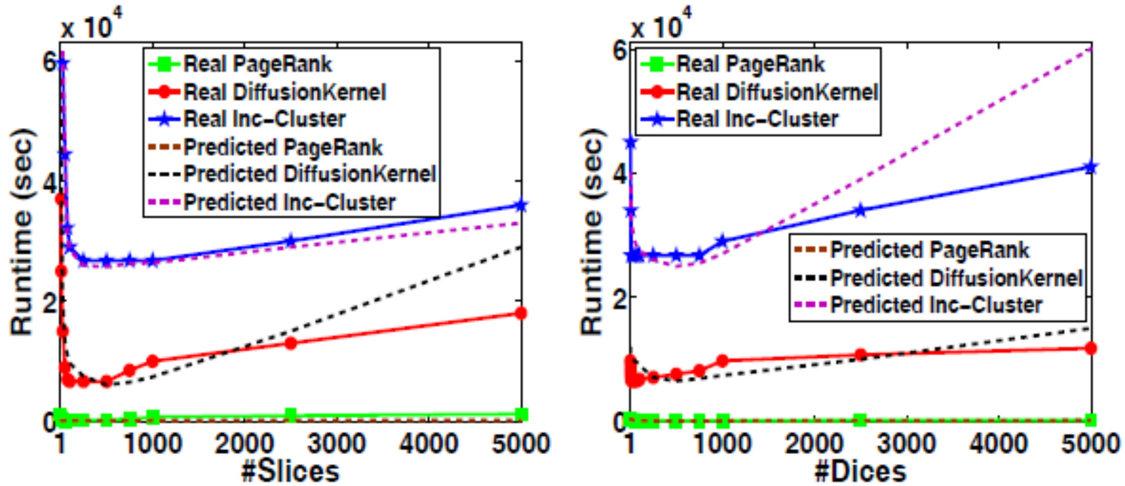


(a) Throughput:#Slices (b) Throughput:#Dices (e) CPU:#Slices (f) CPU:#Dices



(c) Runtime:#Slices (d) Runtime:#Dices (g) Memory:#Slices (h) Memory:#Dices

Efficiency of Regression-based Learning



(a) Different #Slices (b) Different #Dices

Runtime Prediction

	PC (16 GB memory)			PC (2 GB memory)		
Dataset	Facebook	Twitter	Yahoo	Facebook	Twitter	Yahoo
p (#*Slices*)	4	7	13	4	8	9
q (#*Strips*)	3	5	4	4	10	12
r (#*Dices*)	0	4	8	2	7	23

Optimal Partitioning Parameters for PageRank

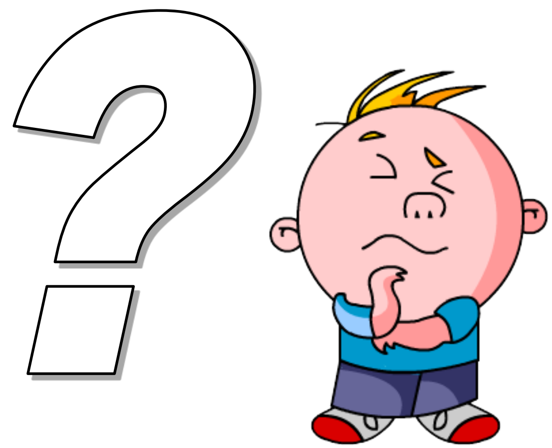
	PageRank			Connected Components		
Dataset	Facebook	Twitter	Yahoo	Facebook	Twitter	Yahoo
p (#*Slices*)	4	7	13	4	6	8
q (#*Strips*)	3	5	4	2	6	7
r (#*Dices*)	0	4	8	0	4	12

Optimal Parameters for PC with 16 GB DRAM

GraphLego: Resource Aware GPS

- **Flexible multi-level hierarchical graph parallel abstractions**
  - Model a large graph as a 3D cube with source vertex, destination vertex and edge weight as the dimensions
  - Partitioning a big graph by: slice, strip, dice based graph partitioning
- **Access Locality Optimization**
  - Dice-based data placement: store a large graph on disk by minimizing non-sequential disk access and enabling more structured in-memory access
  - Construct partition-to-chunk index and vertex-to-partition index to facilitate fast access to slices, strips and dices
  - implement partition-level in-memory gzip compression to optimize disk I/Os
- **Optimization for Partitioning Parameters**
  - Build a regression-based learning model to discover the latent relationship between the number of partitions and the runtime

Questions



Open Source:

https://sites.google.com/site/git_GraphLego/