# GraphMeta: A Graph-Based Engine for Managing Large-Scale HPC Rich Metadata

**Dong Dai** [1], Yong Chen [1], Philip Carns [2], John Jenkins [2],
Wei Zhang [1], and Robert Ross [2]

[1]Computer Science Department, Texas Tech University

[2]Mathematics and Computer Science Division, Argonne National Laboratory

September 11, 2016

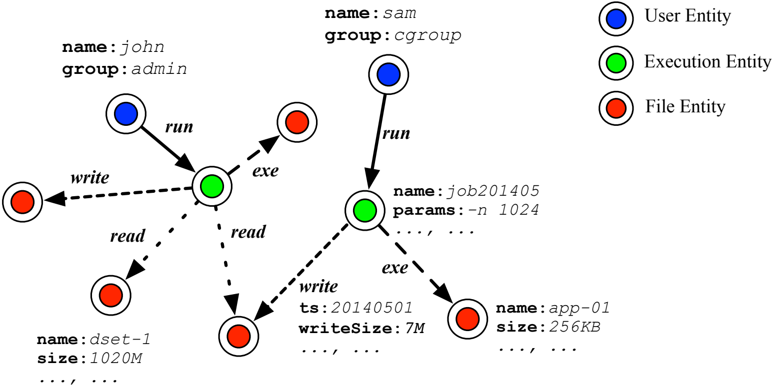
# Introduction: HPC Rich Metadata

High-performance computing (HPC) systems can generate a large amount of metadata about different entities.

- **Traditional Metadata**: POSIX metadata
  - file size/name
  - access permission
  - users and groups
- **Rich Metadata**: metadata about entities like processes, jobs and their relationships with files etc.
  - Well known example: **Provenance**
  - It describes the relationships among entities such as data sources, processing steps, processes, context, and dependencies that contribute to the existence of a data item.
  - Used in various scenes, like data auditing, reproducibility, security, etc.

# Graph-based HPC Rich Metadata Graph

- Rich metadata in HPC systems can be naturally mapped into a property graph[1].



---

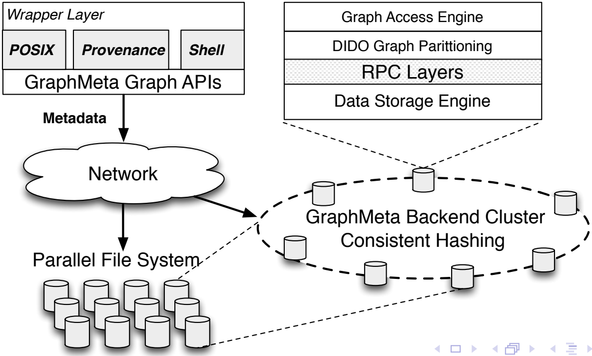
[1] Dong Dai et al. “Using Property Graphs for Rich Metadata Management in HPC Systems”. In: *Parallel Data Storage Workshop (PDSW), 2014 9th*. IEEE. 2014.

# Challenges for Managing Graph-based HPC Rich Metadata

- The challenges of building such a graph-based metadata management solution are significant.
  - **Large data size.** It can have hundreds of millions of vertices/edges;
  - **Large mutation rate.** It needs to store large amount of data/metadata activity from large-scale applications;
  - **Traversal-based access pattern.** Complex graph traversal queries;
  - **Power-law distribution.** Graphs fit the power-law distribution.
- How to support such graph-based HPC rich metadata management?

# GraphMeta Overall Design: Architecture

- Client-Side Component
  - run on compute nodes or login nodes, provide graph APIs and wrappers
- Server-Side Component
  - run on I/O nodes or compute nodes or even a dedicated cluster.
  - graph-partitioning layer; data storage engine; graph access engine
  - cluster organized by consistent hashing; runs on top of existing pfs.



# GraphMeta Overall Design: Data Model

- Property graph data model with **Type**
    - Vertex stores entities.
        - Users can define different types of vertices, each of which has its name and mandatory attributes.
    - Edge stores relations.
        - Users can define different types of edges, each of which can be defined by its name and its source and destination vertices types.
- Property graph data model with **Versioning**
    - An important need of HPC rich metadata is to keep full history.
        - A version is implicitly generated for vertices, edges, and attributes.
    - Global versioning is too expensive for HPC with large mutation rate.
        - Server-side timestamps are used as default versions.
        - Due to time skew, a relaxed session semantics is provided.

# GraphMeta Write-Optimized Data Store

The first goal of GraphMeta:

- high-speed rich metadata ingestion

To satisfy this, we need a write-optimized data store.

- Graphs are mapped to key-value data pairs stored in RocksDB
- All data related with vertex is stored together preserving data locality
- Keys are created following specific rule to enhance the scan speed.

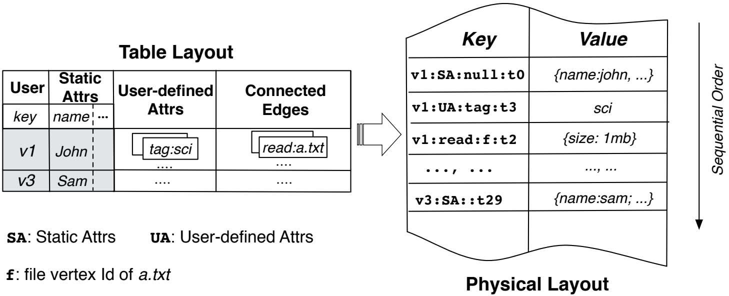


**Table Layout**

| User | Static Attrs | | User-defined Attrs | Connected Edges |
|---|---|---|---|---|
| *key* | *name* | ... | | |
| *v1* | *John* | | *tag:sci*<br>.... | *read:a.txt*<br>.... |
| *v3* | *Sam* | | .... | .... |

**Physical Layout**

| ***Key*** | ***Value*** |
|---|---|
| `v1:SA:null:t0` | *{name:john, ...}* |
| `v1:UA:tag:t3` | *sci* |
| `v1:read:f:t2` | *{size: 1mb}* |
| `..., ...` | *..., ...* |
| `v3:SA::t29` | *{name:sam; ...}* |

`SA`: Static Attrs `UA`: User-defined Attrs

`f`: file vertex Id of *a.txt*

# Graph Partitioning: Motivation and Background

The second goal of GraphMeta:

- to fit large size and mutation rate
- to provide high-performance graph traversal on power-law graphs

This requires/necessitates graph partitioning algorithm:

- Vertices and edges should be distributed to different servers.
- The distribution should be balanced to avoid performance bottleneck.
- Related vertices and edges should be stored together to improve perf.

# Graph Partitioning: Motivation and Background

- Graph partitioning already has been well studied.
- $K$-partitioning for graph $G$.
  - cut $G$ into $k$ balanced pieces
  - minimize the number of edges cut
- Classic offline algorithms.
  - METIS, Chaco, PMRSB, Scotch.
  - They require the global graph structure information.
- Streaming algorithms.
  - LDG, Fennel, restreaming LDG.
  - They need local graph structure information like knowing all connected edges when inserting a vertex
- One-pass/Online Algorithms.
  - Edge-Cut and Vertex-Cut
  - They insert into graphs in an online/one-pass manner, without any graph structure information.

# Graph Partitioning: Performance Analysis

We use 1-step traversal to analysis how graph partitioning affects the graph query/traversal performance:

- The total cost can be defined as follows.

$$T_{scan(v_1)} = T_{v_1} + T_{e_i|e_i \in out\_e(v_1)} + T_{dst(e_i)} \tag{1}$$

- $T_{v_1}$ means the cost of reading $v_1$
- $T_{e_i|e_i \in out\_e(v_1)}$ indicates the cost of iterating all out-edges of $v_1$.

Note: Since $|e_i|$ can be huge in a metadata graph, partitioning them across multiple servers can improve the performance with higher parallelism.

- $T_{dst(e_i)}$, includes the cost of reading all destination vertices of edges.

Note: If $e_i$ and $dst(e_i)$ are partitioned into different servers, extra network communication is needed.

- High-degree parallelism is desired for high-degree vertices.
- Data locality between source/destination vertices and edges is critical.

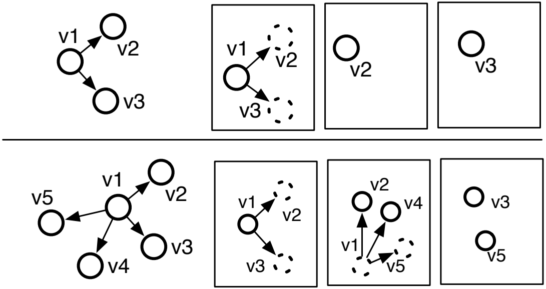
# DIDO Graph Partitioning

We propose an "*destination-dependent optimized* (DIDO)" graph partitioning algorithm to orchestrate parallelism and locality.

1. for better parallelism, DIDO incrementally partitions vertices based on their out-degrees;
2. for better locality, DIDO considers edge placement with the location of respective destination vertices during partitioning.

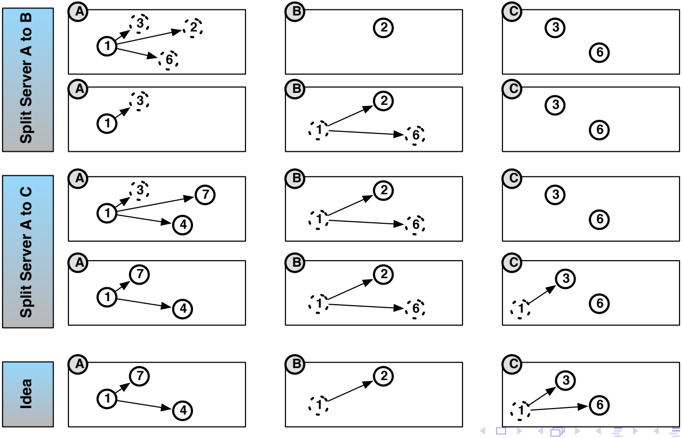
# DIDO - Incremental Part

- Initially, DIDO places a vertex and all its out-edges and associated attributes together on a single server, similar to edge-cut.
  - edges are stored together with their source vertices.
- Once current partition for vertex *v* is having too many edges
  - current partition is incrementally split into two every time its size exceeds the `SPLIT_THRES`.
  - similar to GIGA+



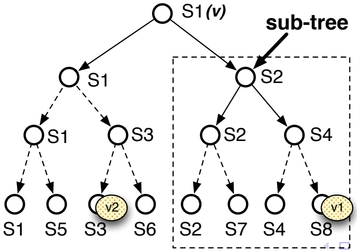
# DIDO - Destination-Aware Part

- Split out-edges in current partition into two: one stays in the original server, and another one is placed to an extended server.
- Which part of current partition should be moved to which server?



# DIDO - Partition Tree

- DIDO relies on *Partition Tree* data structure:
  - the root is $S_v$, the server that stores the source vertex $v$.
  - The left child corresponds to the same server as the parent;
  - the right child indicates the next server not been used in the tree yet.
  - This tree organization is fixed and easy to calculate before any splitting.
- To decide which parts of the out-edges need be moved
  - It calculates the locations of the destination vertices of each edge
  - Always puts edge into the child that leads the path to where the destination vertex is stored.

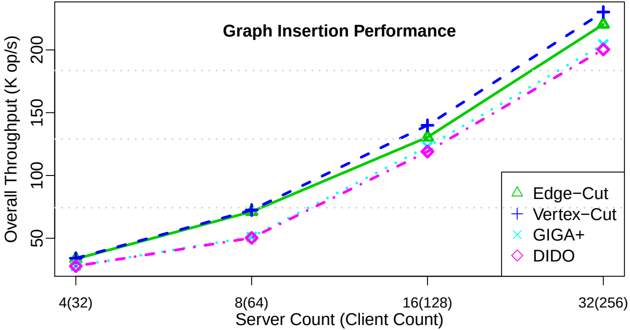


# GraphMeta Evaluation

- Evaluation Hardware:
  - The evaluations were conducted on the Fusion cluster at Argonne National Laboratory.
- Evaluation Datasets:
  - Darshan log generated from a whole year's trace (2013) from the Intrepid supercomputer;
    - The entire graph contains around 70 million vertices and edges, representing the executed jobs, processes, and accessed files, etc.
  - Synthetic graphs, generated by the RMAT graph generator;
    - RMAT parameters: $a = 0.45, b = 0.15, c = 0.15, d = 0.25$

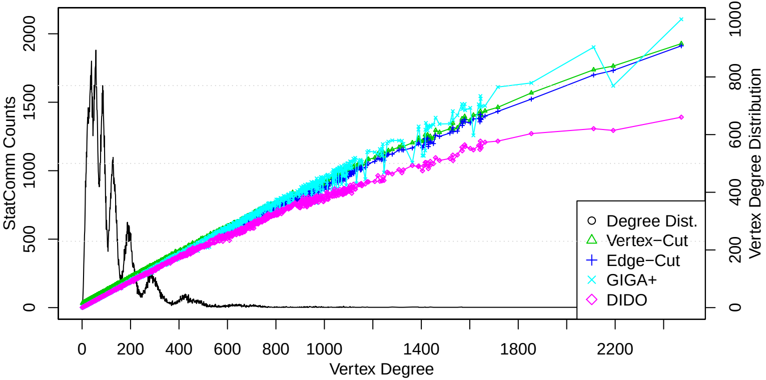
# Evaluating DIDO - Metadata Ingestion Performance

- We compare four graph-partitioning strategies—edge-cut, vertex-cut, GIGA+, and DIDO—for various graph operations
- First, Metadata Ingestion Performance
  - We had $n$ servers and $8 * n$ clients, $n = 4 \rightarrow 32$
  - Each client loaded part of Darshan logs and issued graph insertions in parallel.



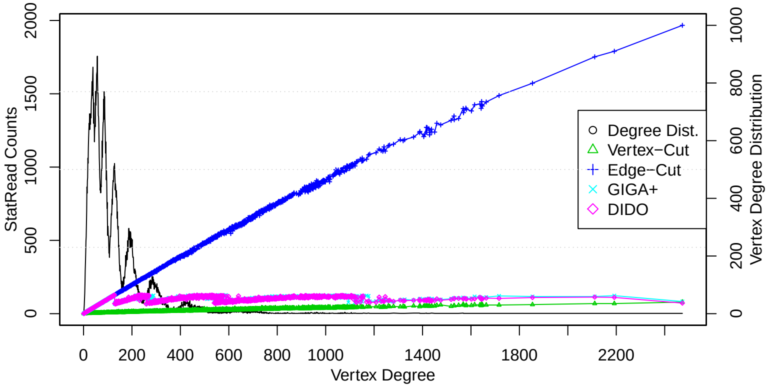
# Evaluating DIDO - Graph Traversal Performance

- Statistical Evaluations
  - **StatComm** measures the cross-server communication.
  - **StatReads** measures the I/O costs across different servers.



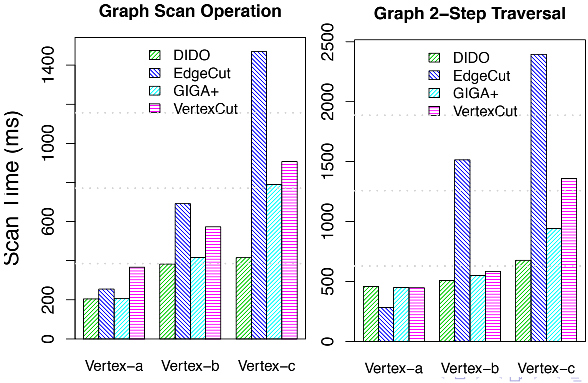
# Evaluating DIDO - Graph Traversal Performance

- Statistical Evaluations
  - **StatComm** measures the cross-server communication.
  - **StatReads** measures the I/O costs across different servers.



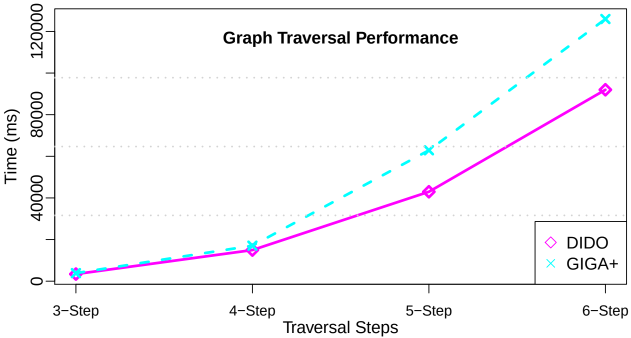
# Evaluating DIDO - Graph Traversal Performance

- Real System Evaluations
  - Run on Darshan dataset
  - Three vertices: *vertex*$_a$ with only 1 edges connected, *vertex*$_b$ with a medium number (572) of degrees, and *vertex*$_c$ with around 10K connected edges.



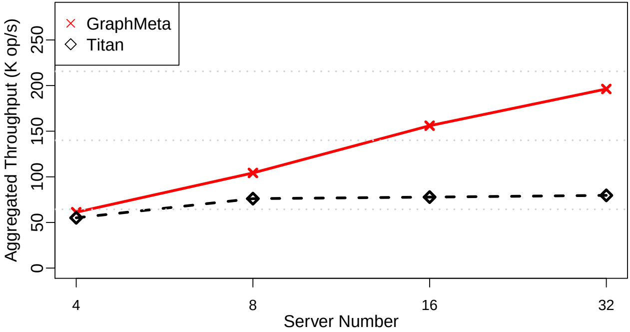
# Evaluating DIDO - Deep Graph Traversal

- The performance benefit of DIDO is shown for scan and 2-step traversal
- In fact, the performance advantage of DIDO can be further confirmed for a longer step traversal due to better locality between edges and their destination vertices.



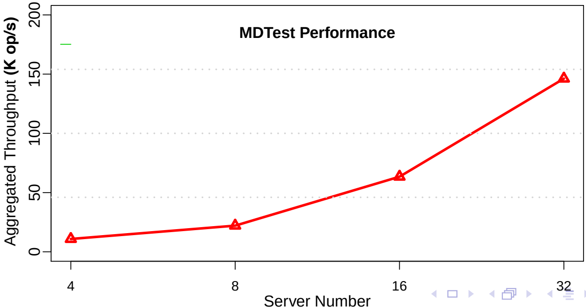
# GraphMeta vs. Graph Databases

- Distributed graph databases can be used for storing and processing rich metadata graphs.
  - manually graph partitioning from clients/users.
  - limited scalability on large-scale power-law graphs.
- We show the performance difference of GraphMeta and a representative graph database, Titan (over Cassandra).



# GraphMeta on POSIX Workloads

- Understand the performance of GraphMeta on POSIX workloads:
  - GraphMeta is not designed to substitute the POSIX metadata service.
  - It needs to keep a valid copy of POSIX metadata for many queries.
- We used *mdtest* benchmark to evaluate the performance of creating large number of files into a single directory.
  - For $n$ servers, $8 * n$ clients issued file creations concurrently.
  - Each client created the same number (4,000) of files.



# Conclusion and Future Work

- We identify the critical challenges on building efficient infrastructure for graph-based HPC rich metadata management
- GraphMeta, with various optimizations, can be used in HPC cases with good performance and scalability
- A new graph partition algorithm, DIDO, is proposed to support the fast on-line one-pass graph partition and provide advantageous traversal performance.

- We plan to investigate fault-tolerance and recovery capability for both graph persistence and complex graph traversal
- we will explore the implementation of a stronger consistency model or, perhaps, transaction support.

# Question and Answer