An Asynchronous Traversal Engine for Graph-Based Rich Metadata Management

Dong Dai[a], Philip Carns[b], Robert B. Ross[b], John Jenkins[b], Nicholas Muirhead[a], Yong Chen[a]

[a]*Computer Science Department, Texas Tech University*
[b]*Mathematics and Computer Science Division, Argonne National Laboratory*

Abstract

Rich metadata in high-performance computing (HPC) systems contains extended information about users, jobs, data files, and their relationships. Property graphs are a promising data model to represent heterogeneous rich metadata flexibly. Specifically, a property graph can use vertices to represent different entities and edges to record the relationships between vertices with unique annotations. The high-volume HPC use case, with millions of entities and relationships, naturally requires an out-of-core distributed property graph database, which must support live updates (to ingest production information in real time), low-latency point queries (for frequent metadata operations such as permission checking), and large-scale traversals (for provenance data mining).

Among these needs, large-scale property graph traversals are particularly challenging for distributed graph storage systems. Most existing graph systems implement a "level-synchronous" breadth-first search algorithm that relies on global synchronization in each traversal step. This performs well in many problem domains; but a rich metadata management system is characterized by imbalanced graphs, long traversal lengths, and concurrent workloads, each of which has the potential to introduce or exacerbate stragglers (i.e., abnormally slow steps or servers in a graph traversal) that lead to low overall throughput for synchronous traversal algorithms. Previous research indicated that the straggler problem can be mitigated by using *asynchronous* traversal algorithms, and many graph-processing frameworks have successfully demonstrated this approach. Such systems require the graph to be loaded into a separate batch-processing framework instead of being iteratively accessed, however.

In this work, we investigate a general asynchronous graph traversal engine that can operate atop a rich metadata graph in its native format. We outline a traversal-aware query language and key optimizations (*traversal-affiliate caching* and *execution merging*) necessary for efficient performance. We further explore the effect of different graph partitioning strategies on the traversal performance for both synchronous and asynchronous traversal engines. Our experiments show that the asynchronous graph traversal engine is more efficient than its synchronous counterpart in the case of HPC rich metadata processing, where more servers are involved and larger traversals are needed. Moreover, the asynchronous traversal engine is more adaptive to different graph partitioning strategies.

Keywords:
Parallel File Systems, Rich Metadata Management, Property Graph, Graph Traversal, Graph Partitioning

1. Introduction

A high-performance computing (HPC) platform commonly generates huge amounts of metadata about different entities including jobs, users, files, and their relationships. Traditional metadata, which describes the predefined attributes of these entities (e.g., file size, name, and permissions), has been well recorded and used in current systems. Rich metadata, which describes the detailed information about entities and their relationships, extends traditional metadata to an in-depth level and can contain arbitrary user-defined attributes. A typical example of rich metadata is provenance or lineage, which maintains a complete history of a dataset, including the processes that generated it, the user who started the processes, and even the environment variables, parameters, and configuration files used during execution [1]. Property graphs, which are an extension of traditional graphs with property annotations on vertices and edges, are a promising data model for rich metadata management in HPC systems because of their ability to represent not only metadata attributes but also the relationships between them. Distributed property graph databases such as

Neo4j [2], DEX [3], OrientDB [4], G-Store [5], and Titan [6] have been developed to assist in managing large property graphs.

We are developing a rich metadata management system based on the new concept of unifying metadata into one generic property graph [1]. In addition to storing the property graphs, a major requirement in the rich metadata use case is to effectively answer graph traversal queries from metadata management utilities, such as provenance queries, hierarchical data traversal, and user audit. Graph traversal usually serves as the basic building block for various algorithms and queries. In fact, it is so fundamental that traversal of simple graphs[1] has been used as a benchmark metric (Graph500) for measuring the performance of supercomputers [7, 8]. Traversal for property graphs is likewise critical and needs efficient implementation.

Typically, the core execution engine of graph traversal is implemented by following the general structure of the parallel "level-synchronous" breadth-first search (BFS) algorithm, dating back three decades [9, 10]. Given a graph G, level-synchronous BFS systematically explores G from a source vertex s level by level. The *level* is the distance or hops it travels. BFS implies that all the vertices at level k from vertex s should be "visited" before vertices at level $k + 1$; hence, global synchronization is needed at the end of each traversal step. The "level-synchronous" breadth-first search structure has been adopted not only in graph databases but also in many distributed graph-processing frameworks, including Pregel [11], Giraph [12], and GraphX [13]. The Bulk Synchronous Parallel (BSP) model is popular in this context because of its simplicity and performance benefits under balanced workload.

However, such global synchronization could cause serious performance problems in our property graph-based metadata management case for several reasons. First, as an on-line database system, our system allows concurrent graph traversals for different management tasks. The interferences among traversals easily create stragglers [14, 15], which can cause poor resource utilization and significant idling during global synchronization. Second, the imbalance of the graph partitions, along with the possible variations in attribute sizes among different vertices and edges, leads to highly uneven loads on different servers (an indication of stragglers) while traversing. The wide existence of small-world graphs in HPC metadata (e.g., degree of vertices follows the power-law distribution [16, 1]) makes this problem even worse. Third, in HPC metadata property graphs, possible graph traversal steps could be much larger than the graph diameter, which traditionally limits the maximal traversal steps in simple graphs. For example, the six degrees of separation theory exists in social networks [17]. Specifically, in our use case, different attributes of the same vertex or edge can be used in different steps. Longer traversals introduce more synchronizations and lead to a higher chance of performance penalty caused by stragglers.

Previous work indicated that asynchronous approaches have the potential to minimize the effects of load imbalance across different cores in multicore machines [18]. GraphLab [19], PowerGraph [20], and other distributed frameworks [21, 22] have investigated the use of asynchronous execution models, which could implement the traversal operations in general. However, these approaches are more suitable for the distributed, batch-oriented graph computation that runs on the entire graph, instead of interactive traversal and query of subgraphs, which are common in our HPC-rich metadata management system.

In this research, we explore the design and implementation of an asynchronous traversal engine. We propose optimizations, including *traversal-affiliate caching* and *execution merging*, to fully exploit the performance advantage of the asynchronous traversal engine. In addition, we explore the effect of different graph-partitioning strategies on graph traversal engines to show the advantage of the asynchronous engine. Also proposed is a general traversal language to describe diverse patterns of property graph-based rich metadata management. We show that the asynchronous engine can support this language with detailed progress report functionality comparable to that of a synchronous engine. The main contributions of this work are fourfold.

- Analysis and summary of the graph traversal patterns in property graph databases for HPC rich metadata management. Based on these patterns, we propose a graph traversal language to support them.
- Design and implementation of an asynchronous distributed traversal engine. Critical optimizations are also proposed for the asynchronous traversal engine: *traversal-affiliate caching* and *execution merging* to improve the performance.

[1] The simple graph indicates a graph defined as a set of nodes connected by weighted edges in this study.

- Analysis of the effects of vertex-cut vs. edge-cut graph partitioning on graph traversal.
- Evaluation and demonstration of the performance benefits compared with synchronous traversal engine on both synthetic graphs and real-world graphs, as well as under different graph-partitioning strategies.

The rest of this paper is organized as follows. Section 2 introduces the background of rich metadata graphs and summarizes their graph traversal patterns by analyzing HPC metadata applications. In Section 3, we introduce the GraphTrek traversal language designed for these patterns, and we show how to use it to implement the given use cases. In Section 4, we describe the asynchronous traversal engine in detail, followed by several optimization strategies in Section 5. Section 6 discusses the effect of different graph-partitioning strategies on traversal performance. In Section 7, we discuss the implementation details of such traversal frameworks. Section 8 presents evaluations, including comparisons with a synchronous implementation and an analysis of the impact of asynchronous traversal optimizations. In Section 9, we present conclusions and discuss future work.

2. Background and Requirements on Metadata Graph Traversal

In this section, we provide background on the metadata graph model and the critical attributes that motivate our design. Then we analyze several use cases of HPC metadata management which can be modeled by using a property graph. Through this analysis, we summarize the graph traversal patterns as the foundation of our proposed language design. The graph traversal requirements also motivate the design and implementation of our proposed asynchronous traversal engine. A more complete discussion and analysis of these use cases can be found in our previous work [1].

2.1. Graph-Based HPC Metadata Management

HPC metadata can be intuitively abstracted as a graphlike structure. For example, metadata—including users, executions of programs (jobs), data files accessed, or simply a directory—can be neatly mapped into different vertices in property graphs, as shown in Fig. 1. For example, *File* represents the basic data unit in storage systems; *Execution* indicates the running applications, such as a *job* submitted by users, *processes* scheduled within one job, or *threads* running inside one process; and *User* means the real user of the cluster. Between these entities, different interactions and relationships can be represented as different types of directed edges with properties attached. For example, the *run* edge indicates that the user started the corresponding execution instance, the *exe* edge denotes which executable file(s) an execution used, and the *read/write* edges indicate the types of operations performed on files from executions. Although all the relationships are shown as directed, the relationships can simply be undirected, relying on vertex attributes to derive meaning. If that is insufficient, there might also be corresponding reverse relationships for traversal in the opposite direction (e.g., a *wasExecutedBy* relationship). Some entity properties are shown for vertices, such as UID/GID, file names, and parameters used by the execution. These properties are by no means exhaustive; and additional properties can easily be added, such as file permissions and creation time.

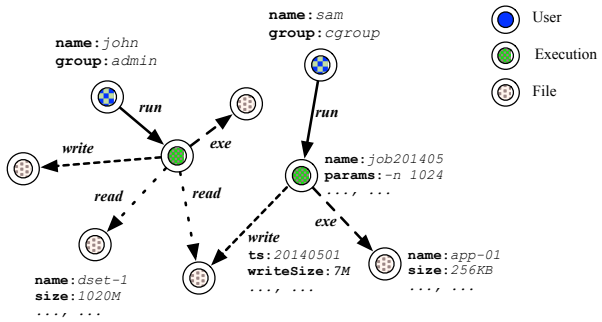

Figure 1: A metadata graph example in HPC Systems.

The graph-based metadata management approach also allows users to define their own entities and map them to new types of vertices. For example, users can create *workflow* entities to connect different *executions* for a workflow system.

2.2. Rich Metadata Graph Attributes

Our previous work examined the structure of graphs representing HPC metadata [1], informing the design challenges of the graph traversal engine we will face. Specifically, we generated graphs from one-year publicly available [23] Darshan I/O logs [24, 25] on the Argonne Intrepid Blue Gene/P supercomputer and analyzed the resulting graph structure, leading to the following key observations.

2.2.1. Large Scale

HPC metadata graphs can have hundreds of millions of vertices and edges, which we expect would increase both with supercomputer scale and with more complex use-cases. In Table 1, the first two columns show the total number of users and jobs in the Darshan trace collection. The column *I/O Ranks* records all ranks (processes) that have I/O operations. In many cases, this indicates the rank 0 process or the aggregators in two-phase I/O. The column *Ranks* records all the ranks as processes whether they performed I/O or not. The column *Files* shows all the files that have been visited by those jobs. This table clearly shows the possible scale of HPC metadata graphs. Hence, graph traversal engines that operate on such graphs must consider scalability and load balancing. In addition, HPC batch jobs typically involve a large amount of data/metadata activity up front and on regular intervals (e.g., checkpointing). A graph traversal engine in this environment must therefore operate safely in the presence of high-volume concurrent updates to the graph and thus most provide a consistent model to determine which data should be returned or discarded.

Table 1: Statistics of Metadata Graph.

	Users	**Jobs**	**I/O Ranks**	**Ranks**	**Files**
Num	177	47,592	10,085,931	113,278,038	34,608,033

2.2.2. Power-Law Distribution

Similar to POSIX file metadata, where a vast majority of directories contain a reasonably small file count while the remaining contain huge counts, rich metadata graphs follow the same pattern with additional unbalanced entities such as users, jobs, and files. In our previous work, we applied the *maximum likelihood estimator* approach [26] to formalize these observations. In that approach, we first assume the sample fits the power-law distribution, then use the maximum likelihood estimator to estimate the possible parameters. With these parameters as a hypothesis, we then use *p-value* to quantify the plausibility of the hypothesis Based on this strategy, we look through the user, process, and file degree distribution again.

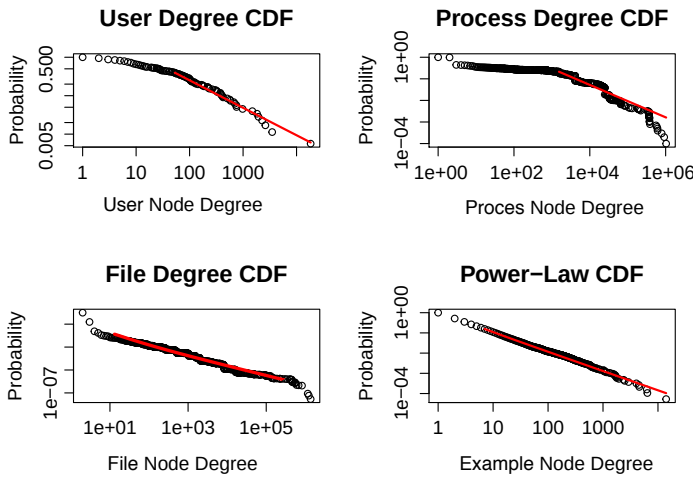

Figure 2: Degree CDF for different entities. The **red line** denotes the power-law distribution.

The results in Fig. 2 show a strong indication that rich metadata graphs, similar to social networks, follow the *skewed power-law distribution* [16]. Specifically, the *User*, *Process*, and *File* entities fit the power-law distribution well. By integrating the results from node degree distribution figures shown in our previous work [1], we assert that the user, process, and file degrees fit the power-law distribution.

The imbalanced nature of metadata graphs presents a challenge for efficient traversal: a single overloaded server may slow down the whole traversal significantly. In addition, the power-law distributed graphs require complex graph partitioning algorithms that in turn increase the complexity of the graph traversal engine.

2.3. Use Cases

In this section we highlight examples of the types of queries that could be performed on rich metadata that is stored by using a property graph data model.

2.3.1. POSIX Filesystem Namespace Management

The hierarchical namespace of POSIX filesystems can be mapped to the graph model by abstracting directories and files as linked vertices. The operations on those files or directories can be mapped to vertex and edge mutations, while operations such as `readdir` can be satisfied by neighbor queries on a particular vertex (a scan). A multilevel directory traversal can be efficiently expressed as a graph traversal. Moreover, the graph-based model can extend the POSIX namespace capabilities by allowing logical organizations of data files as appropriate for the scientific domain. For example, it can allow scientists to simply retrieve all data files generated by a particular experiment.

2.3.2. Data Auditing

Data auditing is critical in large computing facilities where different users share the same cluster. One type of audit query is the following: *Find the set of files read by a specific user during a given timeframe*. Based on the property graph abstraction, this query can be mapped to a graph traversal operation in two steps: (1) beginning at the given user, traverse the edges with the *run* property type to compute the set of executions the user has performed, filtering the results by the given time frame; and (2) traverse the *read* edges from the executions to the resulting files.

2.3.3. Provenance Support

Provenance is widely used in metadata management systems, including data sharing, reproducibility, and workflow management [27, 28, 29, 30]. After modeling rich metadata such as user/file system interactions as property graphs, we can answer provenance questions such as the following: *Find the execution whose model is A and input files have annotation as B* (here, *model* indicates the critical component of the execution, and *annotation* refers to the user-specified attributes on data files). This kind of query, which is a generalized version of a problem from the First Provenance Challenge [31], can be expressed as a graph traversal search that starts from *execution* vertices to *file* vertices while checking needed attributes during the traversal. In contrast to typical graph traversal operations, this query requires the source vertices (*executions*) as the returned value instead of the final *file* vertices.

2.4. Graph-based Metadata Traversal Patterns

Many other use cases have needs similar to those of the cases discussed above. Based on these use cases, we summarize the typical traversal patterns as follows:

- Like BFS, graph traversal starts from a set of vertices and travels in steps. In each step, since it travels through property graphs, it needs to filter the vertices and edges according to attributes.
- Unlike BFS, graph traversal may revisit the same vertex in different steps in order to check different attributes or edges. This kind of revisit is considered as cyclic or redundant in BFS, but it is acceptable in our use case.
- Unlike BFS, graph traversal need not always return the destination vertices. As the provenance example shows, any vertices accessed during the traversal could be needed by users.

Based on these observations, we introduce the proposed traversal language in the following section.

3. Traversal Language

A number of query languages either directly represent or can be used for property graphs [32]. These languages include SPARQL [33] for RDF data [34], GraphGrep for regular expression queries in graphs [35], Cypher for Neo4j graph databases [36], Gremlin from the Thinkerpop project [37], specific query language for provenance [38], Quasar for QMDS [39], and SQL (SemiJoin) for relational databases [40, 41, 42]. Among these languages, previous research [43] suggests that a low-level language such as Gremlin provides better performance because it allows users to manually control each traversal step in detail. But, extremely low-level abstractions—for example, the *vertex-centric* or *edge-centric* graph programming primitives used in distributed graph-processing frameworks such as Giraph and Pregel—place too many implementation burdens on the users, who need to implement their own BFS algorithm using these primitives to finish a traversal. In this research, we propose a more restrictive traversal language that allows users to manually control each step while remaining simple and easy to use.

3.1. Traversal Language API

We define an iterative query-building language to represent property graph traversal operations. The language prototype is implemented in Java. The primary class defined is called GTravel, whose methods return the caller GTravel instance to allow call chaining. The most important core methods are defined as follows; we omit other functions such as *progress report* in this work for brevity.

- *Vertex/Edge selectors v() and e()*

  The vertex selector method *v()* represents an entry point for a graph. These IDs can be initially retrieved with searching or indexing mechanisms provided by any underlying graph storage. The edge selector method *e()* selects specific edges from the working set of vertices (*frontier*) by its label argument, at the point the method call is placed in the call chain.

- *Property filters va() and ea()*

  Property filters take the property key, type of filter, and comparison property values as arguments to filter out vertices and edges. Property key, the name of the property, normally is a byte array. The filter types currently include *EQ*, *IN*, and *RANGE*, which indicate that the given properties of vertices or edges must be *equal to* the value, *within* a set of values, or *in between* the given ranges, respectively. Note that multiple property filters can be applied in one step to filter more entities by using the *AND* operation. *OR* is not explicitly supported in the current version, but users can issue different traversals and combine their results for this purpose.

- *Return indicator rtn()*

  A return method tells the graph traversal engine that the working set of vertices at the point of the call should be returned to the user, but only for those vertices whose resulting traversals reach the end of the call chain. Normally, graph traversals return the final destination vertices or all visited vertices. But, as our HPC provenance example shows, it is useful to be able to return the intermediate results, such as executions whose connected vertices satisfy given conditions. For such traversals, we simply add *rtn()* to the call chain after the traversal step of interest, in order to return the needed vertices.

The graph traversal instance (GTravel) encapsulates multiple steps into a single batch. To start a graph traversal, users will need to build such a GTravel instance by chaining multiple operations sequentially and then submit it to the traversal engine. To join the results from multiple requests, users need to submit them individually and manually combine them in client-side.

3.2. Traversal Commands Applied to Use Cases

Given the graph traversal language, we can easily describe the traversal operations for the use cases discussed in the preceding section.

POSIX File Location. The query *locate a file through a given path* can be expressed as a graph traversal request as follows. It starts from the root directory in POSIX file systems (i.e., "/"), then travel through the "contains" edges to locate all its subdirectories. The file names are used as filters on those vertices during traversal. The destination file will be returned.

```
GTravel.v('/').e('contains').va('name', EQ, 'exp1').e('contains').va('name', EQ, 'input1').rtn()
```

Data Auditing. The query *find all files ending in .txt read by "userA" within a timeframe* can be expressed as follows. First, a vertex selector is used to choose the user vertex of interest. Then, the traversal follows the "run" and "read" edges with property filters applied. As *rtn()* suggests, this command will return the *file* vertices encountered.

```
GTravel.v('userA').e('run').ea('start_ts', RANGE, [t_s, t_e]).e('read').va('type', EQ, 'text').rtn()
```

Provenance Query. The example provenance request *find the execution whose model is A and inputs have annotation as B* is shown below. In this command, we first select all the vertices with given type (i.e., execution) and then denote that these execution vertices are the return vertices using *rtn()*. In this way, the paths that satisfy all these constraints will return their source *execution* vertices to users.

```
GTravel.v().va('type', EQ, 'Execution').rtn().va('model', EQ, 'A').e('read').va('annotation', EQ, 'B')
```

4. Asynchronous Traversal Engine Design

A graph traversal begins from the moment users submit their GTravel instances and ends when users receive all returned vertices. In this section, we introduce the proposed asynchronous traversal execution in detail.

4.1. Execution Engine

The proposed asynchronous traversal execution starts from submitting the GTravel instance in a different way from most existing graph databases (e.g., OrientDB and Titan), in which the traversal is submitted by simply splitting the multistep traversal into multiple queries. Clients issue one query each time and aggregate results to build the next query, as Fig. 3(a) shows. We consider this design as a *client-side* traversal since the client plays a central controller role during the traversal. This design usually leads to performance problems because the clients need all the intermediate results transferred from servers through the busy client-server network. Furthermore, compared with servers, the client is error-prone, thus significantly affecting the system stability.

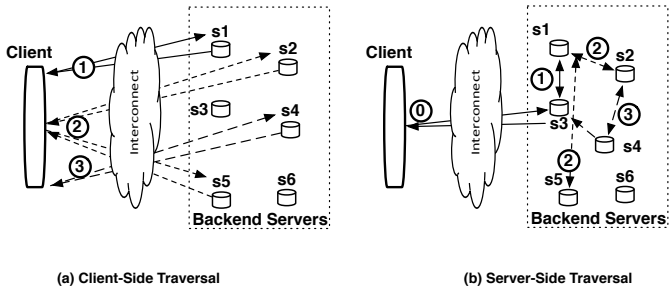

Figure 3: Comparison of client-side traversal and server-side traversal.

In this research, we propose a different strategy, namely, *server-side* traversal. As Fig. 3(b) shows, the client sends the GTravel instance to one selected backend server ($s3$ in this example) to start a graph traversal. This selected server ($s3$) will serve as the *coordinator* for this traversal. The traversal is executed among backend servers and returns the status and results to the *coordinator*. In this way, server-side traversal reduces unnecessary client-server communications, which are typically slower comparing with backend servers. More important, server-side traversal takes advantage of data locality in backend servers hence can provide better performance. This *server-side* traversal is similar to job submission in graph-processing frameworks such as Giraph, Pregel, and GraphX.

The server-side traversal is scheduled upon the coordinator's receipt of the client's GTravel instance. The coordinator develops a multi-step execution plan from the traversal command execution plan and executes it asynchronously, as shown in Fig. 4. Because of the page limit, we omit the property filters on vertices and edges, which in this case are applied locally on each server.

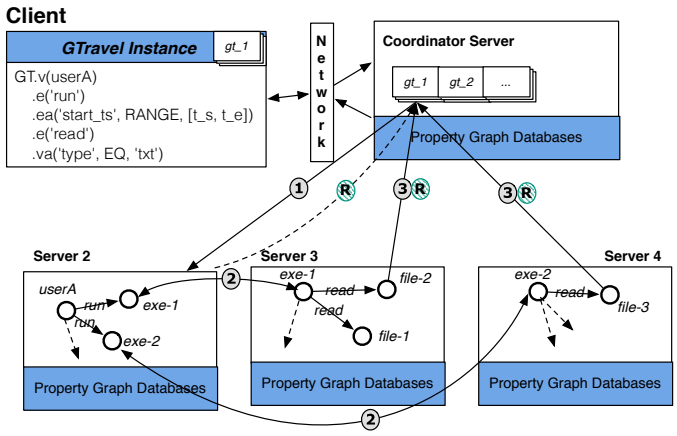

Figure 4: Asynchronous execution of a traversal command. Numbered circles represent the order of operations, circles with numbers represent data (vertex/edge) transfers, and circles with 'R' represent traversal status updates.

In this example, the two-step GTravel instance (gt_1) begins from *userA*. The coordinator server first learns that *userA* is stored in $server_2$ from the underlying graph databases and then sends the request to $server_2$ with an extra parameter specifying the current step *1*. Upon receiving the request, $server_2$ will iterate all the "run" edges from the given vertices and filter them based on specified filter functions. Optimizations can be applied in underlying storage for iterating edges: since we usually iterate edges by type, storing all the edges of one vertex together based on their type will provide better performance for such behavior. With the storage optimizations, the edge iteration on $server_2$ would become sequential, which could obtain the best performance on block-based storage devices.

After iterating edges, we get a new set of vertices, which will be the starting vertices in the second step. Without any synchronization, $server_2$ will concurrently send the GTravel instance (gt_1) to all servers that the vertices are stored at ($server_3$ and $server_4$ in this example). Similarly, the extra parameter that denotes the second step is also attached with the request. Upon receiving this data, $server_3$ and $server_4$ will perform the similar edge iterations to get a new set of vertices and will dispatch requests to more servers. If the current execution is the last step in the traversal command, instead of dispatching the traversal to a further step, the server will return the vertices to the coordinator server, shown as step 3 in Fig. 4. (Returning non-"end" vertices is discussed in Section 4.2.) Once all the vertices are fully returned, the coordinator server starts to reply the client, and the whole graph traversal finishes. A buffered pipeline can be created to transfer results from the coordinator to the clients if the return dataset is too large. We left this optimization as future work.

The asynchronous graph traversal still follows the breadth-first structure: each vertex iterates all its neighbors in parallel, and there is no accessing order among these neighbors. However, different from traditional BFS implementation, which needs synchronization among different steps, we allow each server to start the next step without explicit synchronizing with other servers. Hence, it can coordinate the unnecessary waiting for the slow servers and provides a better overall performance.

4.2. Traversal Return

A traversal stops when it reaches the last step of the GTravel instance. Typically, these final executions will transmit the final vertices to the coordinator and to the clients. But as *rtn()* suggests, we allow users to return the intermediate or even the source vertices. In order to support such functionality, each time that a backend server starts a traversal execution, it will check whether the generated vertices are marked as returned by users. If yes, the server will change the report destination of all the downstream traversal executions to return the needed vertices.

As Fig. 5 shows, changing the "reporting destination" causes the graph traversal to execute in a slightly different way. Assume that the vertices (v_1 and v_2) are generated in a certain step of the traversal, which is marked with *rtn()*. Then, the servers actually storing v_1 and v_2 will force the downstream servers to change their "reporting destination"

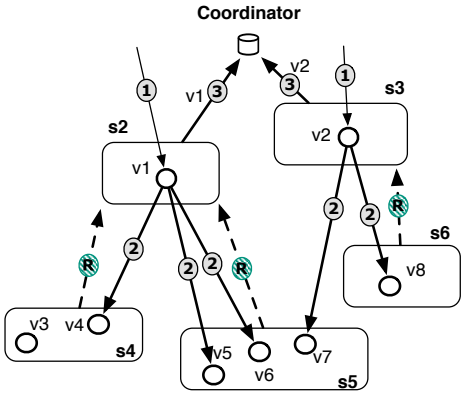

Figure 5: Example of a graph traversal that returns the intermediate vertices.

from *coordinator* to themselves. The servers that execute the last step will send their final results to these reporting destinations. For example, as step *R* shows, after one more step traversal, the backend servers return to $s2$ and $s3$ instead of the coordinator server. When replies arrive, servers ($s2$ and $s3$) will know the status of downstream executions and will send the vertices to the coordinator server accordingly. In this way, we can return vertices in arbitrary steps in the graph traversal.

4.3. Status and Progress Tracing

In asynchronous graph traversal, each server independently executes its actions and spreads the traversal to more servers if needed. No global traversal status can be obtained. This situation introduces a *correctness concern* such as silent failure, which means that if the asynchronous execution fails, the system may not be informed. Because of the existence of such failures, the coordinator server will not be able to decide whether the entire traversal has finished correctly. We introduce a status-tracing mechanism to identify failures to guarantee the traversal correctness. We leave for future work the implementation of full fault tolerance features such as restarting traversal from where it failed.

Consider one backend server as an example. During the traversal, it repeats the same operation: it receives the GTravel instance from another server, performs the needed vertex and edge filtering to get a new set of vertices, then concurrently sends the traversal instance to more servers according to the new set of vertices. We consider this whole procedure on a specific server as one *traversal execution*. An asynchronous graph traversal consists of many such concurrent *traversal executions*. Intuitively, tracing the status of each execution will give us a global view of the traversal. To trace each execution, we log the creation and termination events of executions in the *coordinator* server. If any execution was logged as created but did not terminate (as the result of a timeout or similar reasons), we consider that the server failed.

In Fig. 4, the circles denoted with 'R' show the example tracing reports from the backend servers to the coordinator during graph traversal. Whenever one server successfully sends the GTravel instance to other servers to start the next step, it will report an *execution creation* event to the coordinator telling it that the new execution is created in the target servers. In addition, after the GTravel instances have been successfully sent, the server will report the *execution termination* event denoting its own termination. An execution will not be considered finished in the coordinator unless it has registered all its downstream executions in the coordinator server and has reported its own termination. Similarly, a graph traversal does not finish unless all the executions created are marked as terminated in the coordinator server.

The status reports of the *traversal executions* from the backend servers also help track the traversal progress. Although it is not feasible to have the exact current step of the traversal as it is executed in an asynchronous way, the count of current unfinished *traversal executions* in each step can still help users estimate the remaining work and time.

5. Asynchronous Traversal Optimizations

To achieve even better performance for asynchronous graph traversal, we introduce two critical optimizations: traversal-affiliate caching and execution merging.

5.1. Traversal-Affiliate Caching

One potential drawback of asynchronous traversal is the redundant vertex visit. Unlike the repeated vertex visit, these redundant vertex visits are from the same step on the same vertex and are triggered by asynchronous execution. Figure 6 shows an example of such a scenario: three different paths arrive at v_H in the same step starting from v_a. These three paths (i.e., $a \rightarrow c \rightarrow H$, $a \rightarrow d \rightarrow H$, and $a \rightarrow e \rightarrow H$) go through two servers. Because of the asynchronous execution model, they may arrive at three different times and cause redundant disk I/Os. This drawback wastes precious I/O bandwidth, leading to a performance problem.

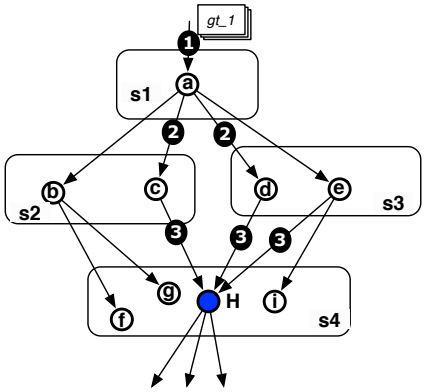

Figure 6: Example of a redundant vertex visit in an asynchronous graph traversal: *a, b, ...* are vertices. *S1, S2, ...,* are backend servers.

To avoid this problem, we introduce a traversal-affiliate cache. In each backend server, a preallocated cache is created once the servers start. During the graph traversal, the server caches the current execution into this buffer with the identification of a {travel-id, current-step, vertex-id} triple. While serving a new request, the server first queries the cache to check whether it has been served before. If there is a cache hit, then the server can safely abandon the request. By doing so, we also avoid conducting graph traversal multiple times.

Although the traversal-affiliate caching buffers only three-element triples, complex and concurrent graph traversal requests can still fill it up. To substitute the cached elements, we use the time-based replacement strategy: for each traversal instance, the triples with the smallest step Ids are substituted. The rationale comes from the fact that the existence of a larger step Id indicates that the oldest steps are already finished. This is still true even under the asynchronous execution model. The distance between the largest step and the smallest step is controlled by using an optimized execution scheduling and merging strategy introduced in next subsection.

5.2. Execution Scheduling and Merging

In the proposed asynchronous graph traversal, each server receives a traversal instance and current step from its ancestor servers. It puts the received requests into a local queue and replies to the ancestor servers before processing these requests. In this way, the ancestor servers can finish asynchronously, and the local server will have a number of buffered requests. A pool of *worker* threads is waiting on this queue for new requests. This leads to a dynamic queue size in each server: if a server is slower or with heavier loads, its internal queue is longer, and more requests are buffered. This presents an opportunity to improve performance via scheduling and merging. Otherwise, the queue is shorter, and the server responds quickly for new requests.

A *worker* thread takes one queued request at a time for processing. The upper queue in Fig. 7 shows an initial status after receiving a number of requests from ancestor servers. We show only a single graph traversal in this figure. During scheduling, the *worker* thread always chooses the request with the smallest step Id in the queue. In this way,

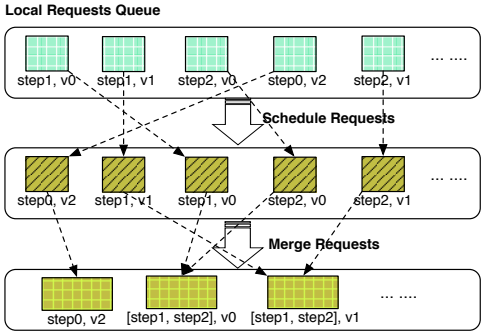

Figure 7: Request queue in local server under scheduling and merging.

all the requests are ordered by their step Ids, as the middle queue in Fig. 7 shows. Using this execution scheduling strategy, we can process the slow steps with higher priority in order to help them catch up. This approach also helps control the maximal step difference between the fastest one and the slowest one in the traversal, thus reducing the traversal-affiliate cache usage.

In addition to execution scheduling, we introduce execution merging. As Fig. 7 shows, we consolidate different steps on the same vertex; for example, traversal executions on $v2$ for step 1 and step 2 will be combined as one disk request. In this way, we need only to retrieve the vertex attributes or to scan its edges once locally. This optimization significantly reduces the amount of disk I/O.

The scheduling and merging on the buffered queue provide an automatic load-balancing mechanism among asynchronous executions inside the same graph traversal. If executions are slower because of stragglers, more requests will be buffered in the queue, providing an opportunity to schedule and merge executions more efficiently. These optimizations can significantly improve the execution of asynchronous graph traversals as confirmed by the evaluation test.

6. Graph Partitioning

Since the metadata graphs are typically distributed across the whole cluster, partitioning the graphs across multiple servers is necessary. Graph partitioning strategies can make a huge difference in the performance of graph traversal. For example, if one vertex with millions of edges is assigned to a single server, traversing through this vertex will be more costly than those low-degree vertices (this can be observed in Section 8). However, partitioning a graph to account for both load balance and vertex-edge locality is difficult. In fact, finding an optimal partitioning strategy for power-law graphs is an NP-complete problem [44], although numerous heuristic algorithms have been developed [45, 46].

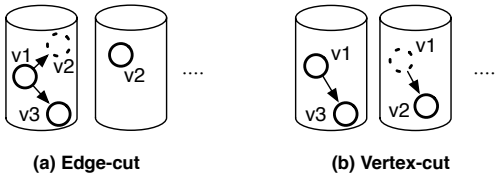

Figure 8: Edge-cut and vertex-cut graph partitioning methods. Solid circles/lines represent vertex/edge storage location, respectively.

Basic graph partitioning strategies include *edge-cut* and *vertex-cut* [20], shown in Figure 8. Edge-cuts distribute vertices, together with their outgoing edges, shown in Figure 8(a). Many distributed graph databases including Titan [6] and OrientDB [4] use this strategy due to both simplicity and vertex/edge locality (i.e., edges are always kept with their source vertices). However, edge-cuts can be problematic for highly imbalanced graphs like a metadata

graph, where vertices can have millions of outgoing edges. On the other hand, *vertex-cut* distributes edges instead of vertices into different servers based on the edge source destination, as Figure 8(b) shows. In this way, the vertices are cut and stored in multiple servers, as its name indicates. This strategy is usually used in graph processing frameworks such as GraphX [13] and PowerGraph[20], where time-consuming computations are applied on each vertex. In this case, *vertex-cut* is proven to have better performance than *edge-cut* [47, 48, 49]. Vertex-cut strategies, however, are inefficient in many cases as they result in unnecessary network communication during traversal for low-degree vertices. Vertex-cut and edge-cut can be considered as two extreme cases optimized for high-degree vertices and low-degree vertices, respectively. Between them, there are a large number of graph partitioning algorithms that cut the graphs adaptively [46, 50, 51, 20].

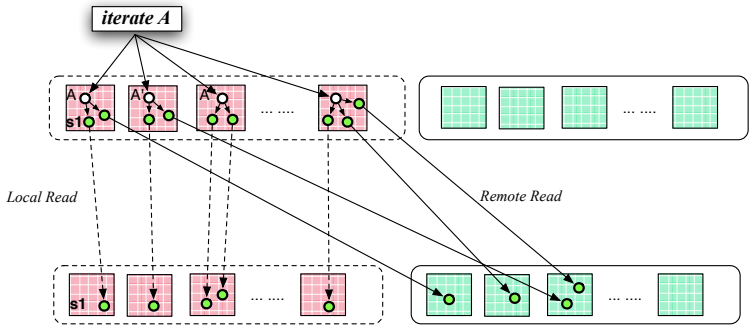

Figure 9: Steps in graph traversal showing effect from graph partitioning.

Graph traversal performance is significantly affected by these partitioning strategies. The basic unit of graph traversal is scanning connected edges of a vertex and broadcasting to more vertices as *iterate A* shown in Fig. 9. The cost of such operation contains broadcasting requests to multiple possible servers that store *A* and its edges plus further reading destination vertices from local or remote servers. Based on this simple model, distributing *A* to more servers increases the broadcasting overhead. On the other hand, aggregating all edges of a vertex together may lead to more remote reading for the destination vertices. The best trade-off clearly is determined by both the network overhead and local read overhead, which are distinct on different clusters. The complexity of traversal performance makes it difficult to analyze the effect of different partitioning algorithms analytically. In addition, the use of asynchronous traversal further complicates such an analysis through overlap of edge scanning and server communication. In this research, we compare those two graph partitioning algorithms through empirical experiments (detailed results are shown in Section 8). Note that, other than the evaluation presented in Section 8, all evaluations are conducted based on edge-cut graph partitions for both synchronous and asynchronous traversal engines.

7. Implementation: GraphTrek

We designed the proposed asynchronous graph traversal engine as a standalone component along with the backend storage system (i.e., the property graph storage system). As Fig. 4 shows, the asynchronous graph traversal engine runs on each backend server independently with the graph databases instances. These traversal engine components communicate with each other through RPC calls but retrieve data and information from the underlying graph storage system using provided APIs. The information that the graph storage provides mainly includes the data and location of a given vertex and edges.

Although numerous distributed property graph storage systems have been proposed and developed, they are all general solutions without considering the requirements of our HPC metadata management use case. For example, Titan stores the property graphs on general column-based NoSQL storage systems such as HBase [52] or Cassandra [53], where all vertices are mapped as different rows; edges and attributes are mapped as separate columns in the same row; key-value storages were also used to implement graph functionalities [54]. Systems such as Noe4j store the graph structure and attributes separately in order to gain performance on queries of graph structure.

In this study, we implemented the proposed traversal engine (namely, *GraphTrek*) and evaluated it based on our own concise but complete graph storage system developed for HPC metadata management.[2] First, it contains client-side portion including libraries providing the graph traversal APIs to support server-side traversal submission. It also has an interactive shell for building and submitting graph traversal queries. Second, the server-side components process graph traversal queries and manage graph distribution and storage. The backend storage system uses consistent hashing to manage the back-end storage cluster by mirroring Dynamo's approach [55]. The mapping from virtual nodes to physical servers is kept in the distributed coordinating service *zookeeper* [56]. The whole storage system runs as a standalone service, similar to IndexFS [57], which runs on I/O nodes or compute nodes using the parallel file system as the storage back-end.

In detail, the graph data such as vertex, edges, and their attributes are organized into different key-value pairs. Those key-value pairs are sequentially stored using RocksDB [58] for better scan performance. The RPC layer is implemented by ZeroMQ [58] as a high-speed network transmission protocol to provide efficient data exchange between graph traversal instances. A graph-partitioning component provides vertex and edge location information to the traversal engine. The graph traversal engine is implemented as described in the previous section.

For comparison, we also implemented synchronous graph traversal in our framework for use as a baseline. In a synchronous graph traversal, a control server typically is used to synchronize each step of the traversal. This control server can be a client or a selected backend server. Using a client as a controller makes the traversal more vulnerable to failures and has worse performance because of multiple rounds of client-server communication. Thus, in our synchronous graph traversal implementation, we follow the same *server-side* traversal design to obtain a fair comparison. The execution of the synchronous traversal is straightforward. Each time, the controller makes sure that all previous executions have finished and then starts the next step. In order to obtain the best performance, the data flows are transferred between involved backend servers without going through the controller. Each server waits for the signal from the controller to start the next step, in order to realize global synchronization between sequential steps.

8. Evaluation

In this section, we evaluate the performance of GraphTrek on synthetic graphs and on a real-world HPC rich metadata management use case. We implemented the synchronous graph traversal (denoted Syn-GT), plain asynchronous traversal without any optimizations (denoted Asyn-GT), and GraphTrek (denoted GraphTrek) for comparison.

All evaluations were conducted on the Fusion cluster at Argonne National Laboratory [59] and CloudLab Utah APT cluster [60]. Fusion contains 320 nodes, and we used 2 to 32 nodes as backend servers in these evaluations. Each node has a dual-socket, quad-core 2.53 GHz Intel Xeon CPU with 36 GB memory and 250 GB local hard disk. Fusion nodes are connected by high-speed network interconnection (InfiniBand QDR 4 GB/s per link, per direction). The global parallel file system includes a 90 TB GPFS file system. CloudLab Utah APT cluster has 128 nodes, and we used 32 nodes as the backend servers. Each node has a 8-core Xeon E5-2450 processor, 16GB RAM and 2 TB local hard disk. CloudLab nodes are connected through 1GbE Dual port embedded NIC. The CloudLab cluster was specifically used to evaluate the performance of different graph partitioning algorithms.

GraphTrek servers are currently able to run using either local storage or parallel file systems by changing the location of RocksDB database files. They can be placed on local disks for better performance or on parallel file systems (e.g., GPFS in the Fusion environment) to provide data resiliency in the case of a server/component failure. In the following evaluations, unless explicitly pointed out, we evaluated GraphTrek atop GPFS. While not presented, the local disk approach improves performance by roughly 10% on Fusion, with the caveat that the resulting service is fault-intolerant.

For experiments with synthetic graphs, we used scale-free graphs generated by the RMAT graph generator [61]. The RMAT graph generator uses a "recursive matrix" model to create graphs that model real-world graphs as social network graphs. We generated directed property graphs with 2^{20} vertices and an average out-degree of 16. The vertices and edges in these synthetic graphs are the same type, with randomly generated attributes attached (the attribute size is 128 bytes). The graph (denoted RMAT-1 graph) was generated with parameters $a = 0.45, b = 0.15, c = 0.15$, and $d = 0.25$, which create a power-law graph with moderate out-degree skewness. In addition to this parameter set, we

[2] GraphMeta Prototype, http://discl.cs.ttu.edu/gitlab/dongdai/graphfs

tried different RMAT configurations during the evaluation. They all generated largely similar results, so we included only this single set because of space limitations.

We execute 2-step, 4-step, and 8-step graph traversal examples starting from the same randomly selected vertex on 2 to 32 backend servers. No extra workloads are generated on any backend servers during the experiments; any workload imbalance is due to the properties of the graph itself. All evaluations are carried out from a cold start in order to force disk access in the traversal engine. Note that the graph size is held constant as we vary the number of servers. In larger-scale experiments, each individual server therefore stores fewer vertices and edges.

8.1. Sensitivity to Asynchronous Traversal Optimizations

We begin our evaluation by investigating the impact of the asynchronous traversal optimizations described in Section 5. GraphTrek introduces two optimizations, traversal-affiliate caching and execution scheduling/merging, to improve the performance of asynchronous graph traversal. To verify the benefits of these optimizations, we evaluate the performance analysis of Sync-GT, ASync-GT, and GraphTrek in a specific case, an 8-step graph traversal on the RMAT-1 graph, as shown in Fig. 10. Here, We Syn-GT denotes the synchronous graph traversal; Asyn-GT indicates plain asynchronous traversal without any optimizations. Other examples are omitted because they show similar results.

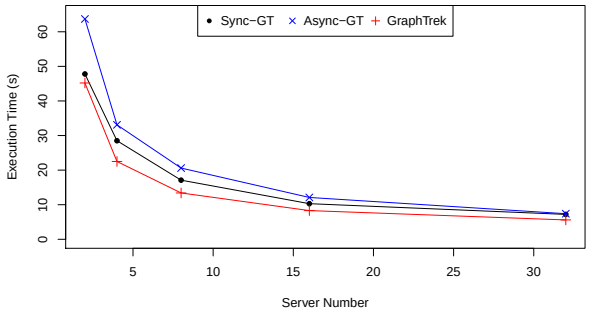

Figure 10: Performance comparison on RMAT-1 graph.

In this series of tests, the Async-GT is the plain asynchronous engine without optimizations. As the results indicate, it has worse performance than that of both the GraphTrek and Sync-GT. The plain asynchronous traversal is even slower than Sync-GT mostly due to the large number of redundant visits. The main performance difference between Async-GT and GraphTrek comes from the proposed optimizations. To further understand the benefits of these optimizations, we placed instruments inside the GraphTrek engine to collect the statistics during the execution. In each server, we collected three statistics: (1) *redundant visits*, which indicates the number of repeated vertex requests detected by the traversal-affiliate caching; (2) *combined visits*, which counts the number of vertex requests that can be combined together by the execution merging; and (3) *real I/O visits*, which counts the real vertex accesses to backend storage systems. The sum of these three numbers equals the total vertex requests received in one server during the traversal. Figure 11 shows the results of a typical run of an 8-step graph traversal on 32 servers (servers are reordered for better presentation).

These statistics show a significant reduction from the received requests to real I/O visits as a result of the proposed optimizations. The redundant vertex visits actually dominate the majority of received requests. Traversal-affiliate caching can effectively remove them in each server, thereby boosting performance. On the other hand, another optimization—execution merging—has different impacts on the different servers: the *combined visit* is much more obvious in the first 10 servers than in the other servers. From an in-depth analysis of these 10 servers, we found that they actually stored more high-degree vertices. Because of this imbalance of graph structure, they were much slower than the other servers while serving the same number of vertex requests. Thus, in an asynchronous engine, the local queue can buffer more requests, and the execution merging was able to merge more vertices together. Therefore, as Fig. 11 shows, these servers end up with fewer real vertex requests and hence can catch up with other servers. This optimization significantly reduces the actual disk I/Os and helps improve the overall performance.

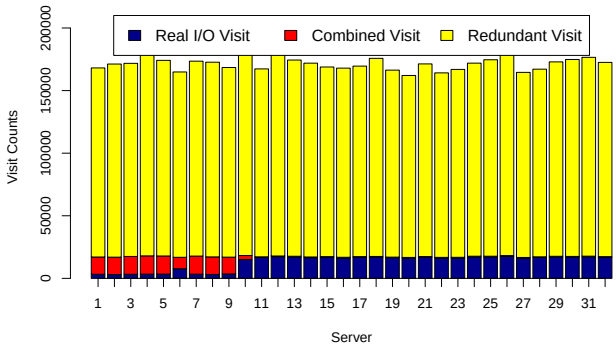

Figure 11: Statistics collected from an 8-step traversal on 32 servers.

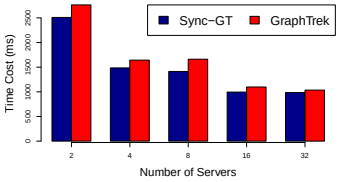

Figure 12: 2-step graph traversal on RMAT-1.

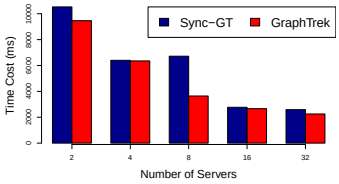

Figure 13: 4-step graph traversal on RMAT-1.

Figure 14: 8-step graph traversal on RMAT-1.

Based on these results, we omit the Async-GT evaluation from all subsequent experiments and focus on the comparison between asynchronous traversal with optimizations (GraphTrek) and synchronous traversal (Sync-GT).

8.2. Performance Comparison on Synthetic Workloads

In this section we use a wider sampling of synthetic workloads. The results obtained by using RMAT-1 are shown in Fig. 12 to Fig. 14. The number of servers is shown on the x-axis, while the y-axis shows the elapsed time (ms) for graph traversal requests.

From these figures we see that for graph traversals with smaller steps and fewer servers, the synchronous implementation actually performs better than does the GraphTrek, as Fig. 12 shows. The reason is that the short traversal does not provide enough optimization opportunities for asynchronous executions. GraphTrek's relative performance improves when more servers are involved in the traversal and the potential for stragglers increases. Figure 13 and Figure 14 also illustrate that GraphTrek performs well with more traversal steps. For example, in Fig. 14, with an 8-step graph traversal, the performance improvement over 32 servers is around 24%, compared with the 5% improvement over 2 servers. The increased number of traversal steps (as would be common in HPC metadata management use cases) also significantly increases the potential for straggler servers to affect performance. With different RMAT graphs, a similar performance pattern can be observed.

8.3. Performance Comparison with External Interference

Servers may experience transient straggling behavior because of concurrent I/O activity from other traversals or external applications. To investigate the performance impact on the traversal engine, we emulated this phenomenon by inserting a fixed (50 ms) delay into individual vertex data accesses. Each time, multiple delays (500 times, indicating 500 slow vertices accesses) were created to emulate a straggler that lasts a certain period of time. By creating fixed delays, we can make sure that the two traversal engines are facing the same amount of external delays. During the whole traversal, three stragglers were created in three selected servers at chosen steps (step 1, 3, and 7). Specifically, we created one straggler chosen by round-robin from these three selected servers in each step. Figure 15 shows how this affected performance for an 8-step RMAT-1 graph traversal.

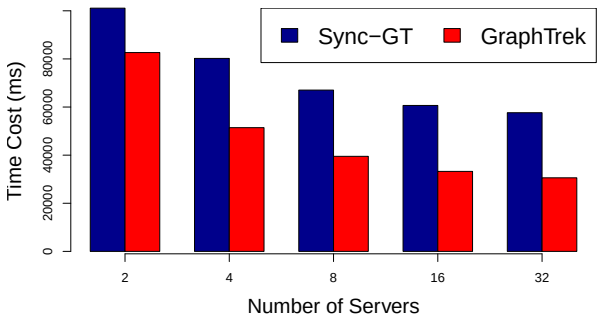

Figure 15: Performance comparison with simulated external stragglers. Each bar shows an average of three runs.

The results suggest an obvious performance advantage of GraphTrek (2x with 32-server) compared with synchronous solutions. The asynchronous traversal can make productive traversal progress despite the presence of external interference because it does not require synchronization after each step of the traversal. The execution scheduling and merging optimizations also allow straggling servers to more quickly catch up with the other servers.

8.4. Performance Comparison with Graph-Partitioning Algorithms

We implemented both vertex-cut and edge-cut graph-partitioning algorithms in order to investigate their impact on traversal performance. The synthetic graph (RMAT-1) is used in this evaluation. To highlight the performance difference caused by different vertex degrees, instead of randomly choosing one vertex to travel, we selected one representative vertex from each vertex degree set and issued traversal requests from it. The results are shown in Fig. 16 and Fig. 17. Here, the maximal degree of generated RMAT-1 graph is 640 and the minimal degree is 1, so the x-axis is in the range of $[1, 640]$. The y-axis shows the value of $\frac{Time(vertex-cut)}{Time(edge-cut)}$, where $Time(x)$ indicates the time consuming of finishing traversal requests for specific graph partition algorithm x. For each start vertex, we issue 1-step, 2-step, and 3-step graph traversal using both synchronous and asynchronous traversal engines. All tests ran on the CloudLab Utah APT cluster with 32 servers storing the graph and one client issuing traversal requests.

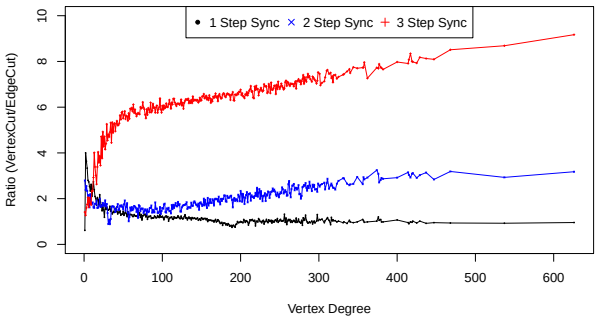

Figure 16: Performance difference of edge-cut and vertex-cut partitioning under synchronous engine.

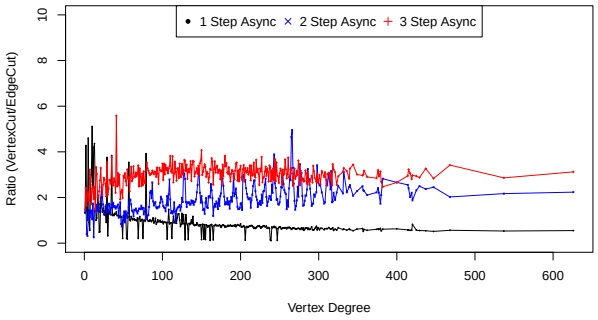

Figure 17: Performance difference of edge-cut and vertex-cut partitioning under asynchronous (GraphTrek) engine.

From Fig. 16 and Fig. 17, we can make several observations. First, vertex-cut consistently performs worse than edge-cut in our test case (i.e., the y-axis values are always larger than 1). The reason is that the largest vertex degree is still relatively small (i.e., around 600) in the test data. The local read cost of iterating through the edges is therefore still relatively low in the edge-cut case when compared with the increased communication cost (32X) introduced by the vertex-cut.

Another important observation is that different partitioning strategies have a significant effect on synchronous traversal performance (up to 10X). However, the effect of different partitioning strategies is much smaller when using

asynchronous traversal as Fig. 17 shows (up to 4X). The asynchronous execution avoids global synchronization, so even when some servers have imbalanced and heavy workloads, they will not affect others and can catch up through merging requests. This indicates that GraphTrek is able to work efficiently under a broader variety of partitioning algorithms.

8.5. HPC Metadata Management Workloads

We also evaluated the impact of the proposed asynchronous graph traversal and the GraphTrek framework for real HPC rich metadata management use cases. To build the heterogeneous property graph, we imported one year of Darshan traces (2013) from the Intrepid supercomputer at Argonne National Laboratory into a property graph for the evaluation [25, 23]. This collection of Darshan logs characterizes the I/O activity of approximately 42% of all core-hours consumed on the Intrepid over the course of a year. Statistics for the generated graph are listed in Table 2. Our previous work shows that this rich metadata graph is also a small-world graph with a power-law distribution [1].

Table 2: Statistics of Rich Metadata Graph

No. of Users	Jobs	Executions	Files	Edges
177	47600	123.4 million	34.6 million	239.8 million

Because of page limit constraint, we show only one example data auditing query and its performance. This query is used for analyzing the influence of a suspicious user on the system. It lists all files that were written by executions whose input files are suspicious. The graph traversal can be expressed as follows.

```
1 GTravel.v(suspectUser).e('run').ea('ts', RANGE, [ts, te])  //select jobs
2     .e('hasExecutions')  //select executions
3     .e('write')          //select outputs
4     .e('readBy')         //select executions
5     .e('write').rtn();   //outputs of executions
```

Running this request for a randomized user on 32 servers with different graph traversal implementations has different performance results, as reported in Table 3. Similar to synthetic graphs, the GraphTrek clearly outperforms the synchronous design and approach.

Table 3: Performance comparison on Darshan graph.

No. Servers	Sync-GT	Async-GT	GraphTrek
32	3575 ms	4159 ms	2839 ms

9. Conclusion and Future Work

Rich metadata is being recognized as increasingly important in future HPC systems in order to support advanced metadata management functionalities. To uniformly manage different types of rich metadata, we proposed a graph-based model and leveraged graph storage systems to efficiently store them. However, existing graph traversal engines are not efficient for metadata graph queries, which are characterized by imbalanced graphs, long traversal lengths, and concurrent workloads. Motivated by the needs of graph-based HPC rich metadata management use cases, we have proposed a graph traversal language to help describe complex queries. We have designed and implemented an asynchronous graph traversal engine called GraphTrek in order to avoid the performance bottleneck caused by stragglers while traveling through rich metadata graphs. To achieve better asynchronous traversal performance, we have also introduced two critical optimizations, traversal-affiliate caching and execution merging, for the asynchronous traversal design. We discussed how graph-partitioning algorithms affect the performance of both synchronous and asynchronous traversal engines. We conducted a detailed comparison of the synchronous and asynchronous traversal engines performance on both synthetic datasets and real-world workloads. The results confirm that for larger systems with deeper traversals, the proposed asynchronous engine achieves better performance than does the traditional synchronous approach. For future work, we will focus on developing fault tolerance capabilities in order to make traversals capable of continuing in the event of faults.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357; and by the National Science Foundation under grant CCF-1409946, CNS-1263183 and CNS-1338078.

Reference

[1] D. Dai, R. B. Ross, P. Carns, D. Kimpe, Y. Chen, Using Property Graphs for Rich Metadata Management in HPC Systems, in: Parallel Data Storage Workshop (PDSW), 2014 9th, IEEE, 2014, pp. 7–12.
[2] J. Webber, A Programmatic Introduction to Neo4j, in: Proceedings of the 3rd annual conference on Systems, Programming, and Applications: Software for Humanity, ACM, 2012, pp. 217–218.
[3] DEX, http://www.sparsity-technologies.com/.
[4] OrientDB, http://www.orientechnologies.com/orient-db.htm.
[5] R. Steinhaus, D. Olteanu, T. Furche, G-Store: A Storage Manager for Graph Data, Ph.D. thesis, Citeseer (2010).
[6] Titan, http://thinkaurelius.github.io/titan/.
[7] D. Bader, K. Madduri, Design and Implementation of The HPCS Graph Analysis Benchmark on Symmetric Multiprocessors, High Performance Computing–HiPC 2005 (2005) 465–476.
[8] The Graph 500 List, http://www.graph500.org/results.
[9] E. Reghbati, D. G. Corneil, Parallel Computations in Graph Theory, SIAM Journal on Computing 7 (2) (1978) 230–237.
[10] M. J. Quinn, N. Deo, Parallel Graph Algorithms, ACM Computing Surveys (CSUR) 16 (3) (1984) 319–348.
[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, 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.
[12] A. Ching, Giraph: Production-Grade Graph Processing Infrastructure for Trillion Edge Graphs, in: ATPESC, ATPESC '14, 2014.
[13] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, GraphX: A Resilient Distributed Graph System on Spark, in: First International Workshop on Graph Data Management Experiences and Systems, 2013.
[14] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, M. Wolf, Managing Variability in the IO Performance of Petascale Storage Systems, in: Proceedings of the 2010 ACM/IEEE SC, IEEE Computer Society, 2010, pp. 1–12.
[15] D. Dai, Y. Chen, D. Kimpe, R. Ross, Two-Choice Randomized Dynamic I/O Scheduler for Object Storage Systems, in: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2014, pp. 635–646.
[16] M. Faloutsos, P. Faloutsos, C. Faloutsos, On Power-Law Relationships of the Internet Topology, in: ACM SIGCOMM Computer Communication Review, Vol. 29, ACM, 1999, pp. 251–262.
[17] Six Degrees of Separation, http://en.wikipedia.org/wiki/Six-degrees-of-separation.
[18] R. Pearce, M. Gokhale, N. M. Amato, Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance computing, networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–11.
[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. Hellerstein, Graphlab: A New Framework for Parallel Machine Learning, arXiv preprint arXiv:1408.2041.
[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, in: OSDI, 2012.
[21] D. Dai, Y. Chen, D. Kimpe, R. Ross, X. Zhou, Domino: An incremental computing framework in cloud with eventual synchronization, in: Proceedings of the 23rd international symposium on High-Performance Parallel and Distributed Computing, ACM, 2014, pp. 291–294.
[22] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, J. Zhong, Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model, in: Proceedings of the 20th ACM PPoPP'15.
[23] Darshan data, ftp://ftp.mcs.anl.gov/pub/darshan/data/.
[24] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley, 24/7 Characterization of Petascale I/O Workloads, in: IEEE International Conference on Cluster Computing and Workshops, CLUSTER'09, pp. 1–10.
[25] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, R. Ross, Understanding and Improving Computational Science Storage Access through Continuous Characterization, ACM Transactions on Storage (TOS) 7 (3) (2011) 8.
[26] A. Clauset, C. R. Shalizi, M. E. Newman, Power-law Distributions in Empirical Data, SIAM Review 51 (4) (2009) 661–703.
[27] P. Buneman, S. Khanna, T. Wang-Chiew, Why and Where: A Characterization of Data Provenance, in: Database Theory ICDT 2001.
[28] D. Dai, Y. Chen, D. Kimpe, R. Ross, Provenance-based Object Storage Prediction Scheme for Scientific Big Data Applications, in: 2014 IEEE International Conference on Big Data, IEEE, 2014.
[29] Y. L. Simmhan, B. Plale, D. Gannon, A Survey of Data Provenance in e-Science, ACM Sigmod Record 34 (3) (2005) 31–36.
[30] C. T. Silva, J. Freire, S. P. Callahan, Provenance for Visualizations: Reproducibility and Beyond, Computing in Science & Engineering 9 (5) (2007) 82–89.
[31] Provenance Challenges, http://twiki.ipaw.info/bin/view/Challenge/.
[32] R. Angles, C. Gutierrez, Survey of Graph Database Models, ACM Computing Surveys (CSUR) 40 (1) (2008) 1.
[33] E. PrudHommeaux, A. Seaborne, et al., SPARQL Query Language for RDF, W3C recommendation 15.
[34] F. Manola, E. Miller, B. McBride, et al., RDF Primer, W3C recommendation 10 (1-107) (2004) 6.
[35] R. Giugno, D. Shasha, Graphgrep: A Fast and Universal Method for Querying Graphsdoi:10.1109/ICPR.2002.1048250.
URL http://dx.doi.org/10.1109/ICPR.2002.1048250
[36] Cypher, http://neo4j.com/developer/cypher-query-language/.

[37] Gremlin, https://gremlindocs.com/.
[38] D. A. Holland, U. J. Braun, D. Maclean, K.-K. Muniswamy-Reddy, M. I. Seltzer, Choosing a Data Model and Query Language for Provenance, in: Proceedings of the 2nd International Provenance and Annotation Workshop (PAWS'08), Springer, 2008.
[39] S. Ames, M. Gokhale, C. Maltzahn, QMDS: A File System Metadata Management Service Supporting a Graph Data Model-based Query Language, International Journal of Parallel, Emergent and Distributed Systems 28 (2) (2013) 159–183.
[40] N. Martinez-Bazan, D. Dominguez-Sal, Using Semijoin Programs to Solve Traversal Queries in Graph Databases, in: Proceedings of Workshop on Graph Data Management Experiences and Systems, 2014.
[41] L. Zou, L. Chen, M. Ozsu, Distance-join: Pattern Match Query in a Large Graph Databasedoi:10.14778/1687627.1687727.
URL http://dx.doi.org/10.14778/1687627.1687727
[42] H. He, A. K. Singh, Graphs-at-a-time: Query Language and Access Methods for Graph Databases, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008.
[43] F. Holzschuher, R. Peinl, Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4jdoi:10.1145/2457317.2457351.
URL http://dx.doi.org/10.1145/2457317.2457351
[44] Ü. i. t. V. Çatalyürek, C. Aykanat, B. Uçar, On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe, SIAM Journal on Scientific Computing 32 (2) (2010) 656–683.
[45] C. E. Tsourakakis, Streaming Graph Partitioning in the Planted Partition Model, arXiv preprint arXiv:1406.7570.
[46] M. Kim, K. S. Candan, SBV-Cut: Vertex-cut Based Graph Partitioning Using Structural Balance Vertices, Data & Knowledge Engineering 72 (2012) 285–303.
[47] A. Abou-Rjeili, G. Karypis, Multilevel Algorithms for Partitioning Power-Law Graphs, in: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), IEEE, 2006, p. 10.
[48] K. Lang, Finding Good Nearly Balanced Cuts in Power Law Graphs, Preprint.
[49] J. Leskovec, K. J. Lang, A. Dasgupta, M. W. Mahoney, Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, Internet Mathematics 6 (1) (2009) 29–123.
[50] N. Jain, G. Liao, T. L. Willke, Graphbuilder: Scalable Graph ETL Framework, in: First International Workshop on Graph Data Management Experiences and Systems, ACM, 2013, p. 4.
[51] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on scientific Computing 20 (1) (1998) 359–392.
[52] A. Khetrapal, V. Ganesh, HBase and Hypertable for Large Scale Distributed Storage Systems, Dept. of Computer Science, Purdue.
[53] A. Lakshman, P. Malik, Cassandra: a Decentralized Structured Storage System, ACM SIGOPS Operating Systems Review.
[54] D. Dai, X. Li, C. Wang, M. Sun, X. Zhou, Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud, in: IEEE Cluster Workshops, 2012.
[55] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's Highly Available Key-Value Store (2007).
[56] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, ZooKeeper: Wait-free Coordination for Internet-scale Systems., in: USENIX Annual Technical Conference, Vol. 8, 2010, p. 9.
[57] K. Ren, Q. Zheng, S. Patil, G. Gibson, IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, IEEE, 2014, pp. 237–248.
[58] RocksDB, http://rocksdb.org/.
[59] Fusion Cluster, http://www.lcrc.anl.gov/fusion/.
[60] CloudLab, https://www.cloudlab.us/.
[61] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A Recursive Model for Graph Mining, in: Proceedings of the 2004 SIAM International Conference on Data Mining, Vol. 4, SIAM, 2004, pp. 442–446.