A Highly Efficient Runtime and Graph Library for Large Scale Graph Analytics

Ilie Tanase[1], Yinglong Xia[1], Lifeng Nai[2], Yanbin Liu[1], Wei Tan[1]
Jason Crawford[1], and Ching-Yung Lin[1]
[1]IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA
[2]Georgia Institute of Technology, Atlanta, GA 30332, USA
{igtanase,yxia,ygliu,wtan,ccjason,chingyung}@us.ibm.com lnai3@gatech.edu

ABSTRACT

Graph analytics on big data is currently a very active area of research in both industry and academia. To support graph analytics efficiently a large number of graph processing systems have emerged targeting various perspectives of a graph application such as in memory and on disk representations, persistent storage, database capability, runtimes and execution models for exploiting parallelism, etc.

In this paper we discuss a novel graph processing system called System G Native Store which allows for efficient graph data organization and processing on modern computing architectures. In particular we describe a runtime designed to exploit multiple levels of parallelism and a generic infrastructure that allows users to express graphs with various in memory and persistent storage properties. We experimentally show the efficiency of System G Native Store for processing graph queries on state-of-the-art platforms.

1. INTRODUCTION

Large scale graph analytics is a very active area of research in both academia and industry. Researchers have found high impact applications in a wide variety of Big Data domains, ranging from social media analysis, recommendation systems, and insider threat detection, to medical diagnosis and protein interactions. These applications often handle a vast collection of entities with various relationship, which are naturally represented by the graph datastructure. A graph datastructure $G(V, E)$ consists of a set of vertices V and a set of edges E representing relations between various vertices.

The design of a graph processing system (GPS) includes two major components: the graph data structures and the programming model. In this paper we discuss the design decisions we employed in System G Native Store to create a flexible graph library that can be tuned and customized by users based on their application specific needs. For the graph data structure some of the major decisions to be made are:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
GRADES'14, June 22 - 27 2014, Snowbird, UT, USA
Copyright 2014 ACM 978-1-4503-2982-8/14/06$15.00
http://dx.doi.org/10.1145/2621934.2621945.

- Is the graph stored in memory, on disk, or both?
- Is the graph directed or undirected ?
- Is the graph sparse or dense ?
- Are there properties associated with vertices and edges?
- Are ACID properties supported?

Depending on a particular problem to be solved, the answers to these questions vary, and often the more functionality that is required, the more complicated the design becomes. Once the data structure is decided upon, the next big decision is selecting the programming model exposed to users. In addition to the datastructure API [3] the programming model may include support for graph algorithms, expressing parallelism on shared memory machines or large clusters, and support for fault tolerance. As part of the programming model users are provided with an interface that can vary from a full runtime system [9][2][8] to a specialized query language[11][1].

In the rest of this paper we answer the questions we raised in this section in the context of a novel graph processing solution called System G Native Store. In Section 2 we discuss related work, Section 3 provides more details of System G, Section 4 discusses our graph datastructure design decisions and Section 5 presents the runtime system. We evaluate the performance of our GPS in Section 6 and conclude in Section 7.

2. BACKGROUND AND RELATED WORK

In this section we discuss related projects and we classify them into three broad categories: graph datastructure libraries, graph processing frameworks, where the emphasis is on the programming models, and graph databases, where the focus is on storage.

Graph libraries: Graph libraries for in-memory-only processing have been available for a long time. For example BOOST Graph library (BGL) [12] provides a generic graph library where users can customize multiple aspects of the datastructure including directness, in memory storage, and vertex and edge properties. This flexibility gives users great power in customizing the datastructure for their particular needs. Parallel BOOST graph library [6], STAPL [7] and Galois [10], provide in memory parallel graph datastructures. All these projects provide generic algorithms to access all vertices and edges, possibly in parallel, without knowledge of the underlying in-memory storage implementation. System G Native Store employs a similar design philosophy with these libraries but it extends these works with support for persistent storage and a flexible runtime for better work

scheduling.

Graph processing frameworks: Pregel and Giraph [9, 2] employs a parallel programming model called Bulk Synchronous Parallel (BSP) where the computation consists of a sequence of iterations. In each iteration, the framework invokes a user-defined function for each vertex in parallel. This function usually reads messages sent to this vertex from last iteration, sends messages to other vertices that will be processed at the next iteration, and modifies the state of this vertex and its outgoing edges. GraphLab [8] is a parallel programming and computation framework targeted for sparse data and iterative graph algorithms. Pregel/Giraph and GraphLab are good at processing sparse data with local dependencies using iterative algorithms. However they are not designed to answer ad hoc queries and process graph with rich properties.

TinkerPop [3] is an open-source graph ecosystem consisting of key interfaces/tools needed in the graph processing space including the property graph model (Blueprints), data flow (Pipes), graph traversal and manipulation (Gremlin), graph-object mapping (Frames), graph algorithms (Furnace) and graph server (Rexster). Interfaces defined by TinkerPop are becoming popular in the graph community. As an example, Titan [4] adheres to a lot of APIs defined by TinkerPop and uses data stores such as HBase and Cassandra as the scale-out persistent layer. TinkerPop focuses on defining data exchange formats, protocols and APIs, rather than offering a software with good performance.

Graph stores: Neo4J [11] provides a disk-based, pointer-chasing graph storage model. It stores graph vertices and edges in a denormalized, fixed-length structure and uses pointer-chasing instead of index-based method to visit them. By this means, it avoids index access and provides better graph traversal performance than disk-based RDBMS implementations.

3. SYSTEM G GRAPH DATABASE OVERVIEW

Comprehensive graph processing solution: System G Graph Database has been actively developed by IBM, where G stands for graph. It provides a whole spectrum graph solution shown in Figure 1, covering graph visualization, analytics, runtimes, and storage. Although System G Graph Database includes a version based on HBase/HDFS [5] and an interface using DB2/DB2RDF or TinkerPop compatible DBs, in this paper we focus on a novel version called System G *Native Store*, which is independent of any existing product or open source software.

High performance graph runtime: System G Graph Database provides a generic C++ sequential runtime library, a concurrent runtime library for multithreading computing platforms, and a distributed runtime library for computer clusters. For distributed library, it offers an efficient communication layer implemented by the IBM Parallel Active Message Inference (PAMI) and the Remote Direct Memory Access (RDMA). All libraries are highly optimized towards state-of-the-art computer architectures such as POWER multicore. Additional graph computing accelerators based on FPGA and GPU are reserved for exploring highly parallel implementation for some graph computing primitives.

Native Graph Storage: Despite of several storage supported by System G Graph Database shown in Figure 1, we focus on the Native Store for graphs. This store persists a graph by saving its structure and properties on both vertices and edges into a set of files. Since it is a file-based store implemented using C++, it is not only highly efficient, but also very portable to various file systems and storage hardware. The graph structure and properties are stored separately, and the properties can be further divided into subgroups. Thus, it avoids loading unnecessary data for analytics. The Native Store also supports multiple versioning at the vertex level, which is the basis for the bi-temp feature and transaction management in database.

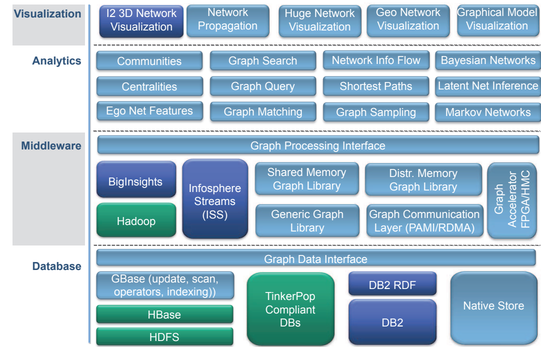

Figure 1: System G Graph Database provides whole spectrum solution for graph processing

4. GENERIC GRAPH DATA STRUCTURE

A graph consists of a collection of vertices and edges. In this section we discuss the functionality our graph library supports as we answer the questions raised in the introduction. Our in memory graph representation uses the adjacency list to store vertices and edges. Thus, the graph stores a set of vertices and each vertex individually maintains the list of its neighboring edges or its adjacency. We have chosen the adjacency list versus either representations like Compressed Sparse Row (CSR), list of edges or adjacency matrix due to its high flexibility in accommodating dynamic graphs where vertices and edges are added and removed while simultaneous supporting traversals.

Similar to BGL or STAPL our framework employs a generic C++ design using templates to allow users to customize a particular graph datastructure. The core graph data structure models a single property graph. In this model the graph stores one property for each vertex and one property for each edge as shown in Figure 2, Lines 1-5.

Users can define different graph classes by appropriately customizing any of the available template arguments. The first template argument is the vertex property; the second template argument is the edge property; the third argument specifies if the graph is directed, undirected or directed with predecessors; the last template argument allows for fine, low level customizations related to the graph storage like the particular storage datastructure for vertices and edges.

We selected a generic programming design as it provides our users the polymorphism flexibility without the runtime overhead of virtual inheritance. Independent of the template arguments used to instantiate it, the graph provides an interface to add and delete vertices and edges and to access the data. The interface is similar to Tinkerpop BluePrints API.

The most important methods of the SG graph class are

```
template <class VertexProperty,
              class EdgeProperty,
              class DIRECTEDNESS,
              class Traits>
class Graph {
  typedef ... vertex_descriptor;
  typedef ... edge_descriptor;
  typedef ... vertex_iterator;
  typedef ... edge_iterator;
  vertex_descriptor add_vertex(VertexProperty&);
  edge_descriptor add_edge(vertex_descriptor v1,
                           vertex_descriptor v2,
                           edge_property&);
  vertex_iterator find_vertex(vertex_descriptor);
}

typedef Graph<int,double,DIRECTED> graph1_type;

class my_vertex_property {...}
typedef Graph<my_vertex_property, double,
              UNDIRECTED> user_graph2_type;

template <class G>
process_vertex(G& g, vertex_descriptor vid){
  vertex_iterator vit = g.find_vertex(vid);
  //process vertex property
  // vit->property();
  edge_iterator eit = vit->edges_begin();
  for (;eit != vit->edges_end();++eit){
    //process edge identified by eit
    // eit->target(); eit->property();...
  }
}
```

Figure 2: System G Native Store Graph Interface

add_vertex, add_edge, and find_vertex. `add_vertex` shown in Figure 2, Line 10 creates a new vertex in the graph and returns the vertex identifier associated with it. The vertex identifier can be used to access and modify the vertex such as adding edges to the vertex, obtaining an iterator (pointer) to the vertex to inspect its properties or its adjacency list, etc. The method `add_edge` shown in Figure 2, Line 11, creates a new edge in the graph between two vertices and with a given initial property. Line 14 shows `find_vertex` which returns an iterator pointing to the vertex data structure. The interface includes additional methods that can't be all described here due to the lack of space.

4.1 Vertex processing

In Figure 2, lines 23-33 we show a simple example of a vertex based computation. First, a vertex is always identified by a unique vertex identifier and the first step to be done before querying properties of the vertex is to map from the vertex identifier to a vertex reference (vertex iterator). Having access to a vertex iterator, one can access the vertex properties (Figure 2, line 27) and information about the adjacency list as shown in Figure 2, lines 28, 29. Traversing all outgoing edges of a vertex can be accomplished with the loop in Figure 2, lines 28-33 and using an edge iterator one can access information about a particular edge like source, target and edge property. All other graph computations in our framework can be expressed as compositions of the pattern included in this example.

4.2 Native Store graph classes hierarchy

The generic graph datastructure introduced in previous section provides enough functionality for a large category of in-memory graph algorithms and analytics. In this section we introduce two more important extensions of our core graph datastructure: multiproperty graph and graph with persistent storage as shown in Figure 3.

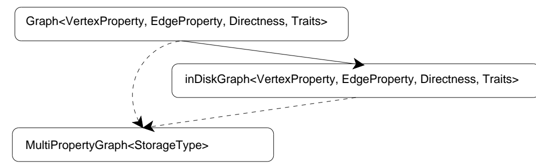

Figure 3: Native Store class hierarchy. The base graph class is in memory only. The inDiskGraph derives from it and adds storage capability. The multiproperty graph can derive from either base graph or inDiskGraph as requested by user.

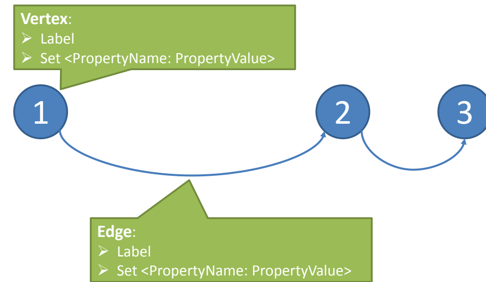

Figure 4: Multi property graph

4.3 Multiproperty graph

A common graph used for graph analytics is the multiproperty graph where each vertex and edge can have an arbitrary number of properties. This functionality is provided by our graph framework by instantiating the base graph class with a multiproperty class for both vertices and edges. A multiproperty class is essentially a map data structure allowing dynamic insertion and deletion of an arbitrary number of key, value pairs.

In Figure 4 we show a depiction of a multiproperty graph. Each vertex has associated a vertex *typeid* which is a mechanism of grouping the vertices based on similarity. For example in a social network graph where we have people, forums, and posts as vertices one can associate a typeid with each group of vertices and later perform analytics over vertices of a particular typeid. In addition to the typeid each vertex can store an arbitrary number of key, value pairs that we will refer to as property names and property values. Similarly for edges we associate an edge label and an arbitrary number of property, value pairs. Additionally the multiproperty graph is directed, but it does keep track of the predecessors for each vertex. Thus one can iterate efficiently over outgoing and incoming edges of a particular vertex.

In addition to the multiproperty support already mentioned a property graph provides an additional interface to the user for allowing traversal of vertices and edges of a particular typeid or label, and additional support for indexing. For example if for every vertex there is a property called "LastName", one may want to search a vertex where "LastName" has a particular value. The multiproperty graph allows users to specify an index on a particular property name and subsequent operations on graph will have to maintain

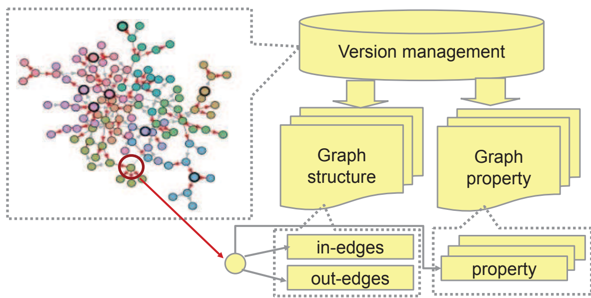

Figure 5: Property graph persistency in System G Native Store

the index and use it for faster lookups.

4.4 Persistent graph

System G provides persistent storage for property graphs as a natural extension of the in-memory graph data structure. Compared to existing graph databases, this graph storage is featured by its analytic amenable design. The basic idea of the storage is shown in Figure 5, which can be described as follows:

`Separating structure and property`: System G stores graph structure and properties separately. The graph structure breaks down into the adjacent incoming edges and outgoing edges for each vertex. Such separation avoids loading graph properties for analytics defined merely on graph topology, e.g. breadth first search (BFS) and graph betweenness. This not only reduces disk IO consumption but also critically improves memory utilization and reduces partition count for large shared-memory graphs.

`Efficient graph loading`: System G allows loading a graph *on demand*, where we start with an empty graph in memory and incrementally load the vertices and edges as needed. This feature allows us to launch a graph analytics fast and also handle graphs larger than the size of memory. System G uses the internal vertex IDs and edge IDs as offsets to store the graph data on disk. Regardless of the size of the graph, it locates the vertex data immediately without incurring any scan or searching. In addition, it is worth noting that it stores adjacent edges to a vertex contiguously, unlike some existing systems e.g. Neo4j. Therefore, System G achieves improved data locality.

`Versioning support`: System G persistent graphs associate time stamps with each vertex. Therefore, it allows transactions and rollback in graph operations, which is critical in many real applications.

`Portability`: System G persistent graph utilizes a set of files to store graph data and therefore can be virtually hosted in most file systems and random access hardware, such as regular Linux file system and HDFS on HDD or SSD.

4.5 Concurrent and Distributed Graph

To fully exploit the large number of threads available on most modern machines, we provide a multi-threaded graph (MTG) data structure within our framework. We take a compositional approach for this. A MTG is composed of an arbitrary number of nearly-independent subgraphs. This approach allows us to maximize coarse grain parallelism.

However, we also enable each individual thread to access any of the subgraphs and this may lead to unsafe concurrent access from multiple threads. The MTG data structure provide atomicity for all of its methods and in the current implementation only one thread can operate on a subgraph at a given point of time. For example building an MTG using multiple threads is efficiently done as each thread can potentially add and access vertices from independent subgraphs thus leading to no conflicts. If the application leads to situations where two threads access the same vertex or edge then the access will be serialized using locking.

Similar to MTG we provide a distributed memory graph (DISTG). A DISTG is thus composed of an arbitrary number of subgraphs, with one or more subgraphs on each machine. The runtime of our library provides remote procedure call (RPC) as the main modality of accessing remote data or performing certain computations on remote data and eventually returning results. The distributed memory graph is currently work in progress that will be deployed in our library relatively soon.

4.6 Java interoperability

Native Store supports Java clients through an in-process JNI layer that maps the concepts and methods of the multiproperty C++ graph to Java static methods. This has allowed the System G team to leverage existing open source Java-based code bases to implement additional graph features. As a result, System G users have the ability to access their graphs through Groovy, Gremlin and SPARQL. In most cases Java clients do not program to the JNI interface but instead program to a TinkerPop Blueprints layer built upon the JNI methods.

5. PROGRAMMING MODEL

In Section 4 we had an overview of the main interfaces of the graph data structures available in System G Native Store. In this section we present in more detail the programming model exposed to the users.

5.1 Single thread programming

The graph exposed by our library is a two dimensional data structure consisting of a set of vertices and a set of edges. We employ an adjacency list and the consequence of this is that we don't store the edges as a contiguous set but each vertex stores its outgoing edges. Thus traversing the whole structure of a graph is often a composition of two nested loops: a first one over each vertex, and a nested one over each edge of a vertex. This is exemplified in Figure 6. If the user requires some analysis over vertices and edges of particular labels then the label needs to be provided as argument to begin and end methods.

If the user performs a local query starting at a particular vertex identified by its vertex descriptor, the code as shown in Figure 2, lines 23-34, can be used.

5.2 Support for concurrent execution

Most parallel machines available today employ multicores and System G Native Store library provides the necessary support to exploit multiple threads. Native Store runtime is the component of our framework that exposes to developers a task based model of computation isolating them from notions like posix threads, cores, SMT, etc.

A parallel computation in Native Store consists of a set of tasks with possible dependencies between them. The set of tasks for a particular computation in general is decided

```
1  template <class G>
2  void process_graph(G& g){
3    vertex_iterator vit= g.vertices_begin();
4    for(;vit !=g.vertices_end();++vit){
5      edge_iterator eit = vit->edges_begin();
6      for (;eit != vit->edges_end();++eit){
7        //process edge identified by eit
8        // eit->target(); eit->property();...
9      }
10   }
11 }
```

Figure 6: Native Store whole graph traversal

```
1  template <class Graph>
2  class addverts_wf : public ibmppl::work_function {
3   virtual void execute(task_id){
4   }
5  };
6
7  //Case 1 : populate graph with vertices
8  addverts_wf<Graph> av(g);
9  ibmppl::execute_tasks(&av, 2 * num_threads);
10
11 //Case 2 : perform a computation on each vertex
12 ibmppl::for_each(g, process_vertex);
13
14 //Case 3 : perform a computation for each vertex of
15 // the task graph
16 ibmppl::task_graph tg;
17 . add vertices(tasks) and edges (dependencies) to tg
18 ibmppl::schedule_task_graph(tg);
```

Figure 7: Expressing parallelism in System G Native Store

when the computation is started (static computation). However there are situations when tasks can be created dynamically by other tasks. This situation is accommodated by our framework with the only complication being the internal mechanism to track down the number of task created and completed in order to let the user know when the computation is finished. Tasks created after the start of computation are also managed by the runtime scheduler which employs work stealing to balance the computations between threads. For performance reasons work stealing is important for such system where the amount of work per vertex is variable as it often depends on how many successors, predecessors the vertex has.

On top of the task based execution model we implemented in System G Native Store a set of primitives to abstract the parallelism from the user. In Figure 7, Lines 1-9, we show a simple example on how users can specify a work function which is the body of the task, and how it can schedule a number of tasks that is a multiple of the number of cores on the machine. The particular code in this figure can be used for example to populate the graph with vertices and edges. If the graph already has data, then the number of the tasks created can vary from the number of cores to the number of vertices in the graph depending on the granularity of the computation per vertex. Using `execute_tasks()` we don't specify a partition of data per task and we leave that to the user.

A second primitive we provide as part of our runtime is the `for_each()`, shown in Figure 7, Line 11. In this case the users is requesting the work function `process_vertex` depicted in Figure 2 to be invoked on each vertex of the graph. The framework underneath decides how to create tasks to maximize the throughput of vertices processed per second.

A last primitive supported by Native Store runtime is `schedule_task_graph()` exemplified in Figure 7, Lines 14-17. In this case the user provides as input a task graph which is a directed acyclic graph. Each vertex represent a task identified by its unique task identifier and its associated work function. The edges represent dependencies that will be enforced when processing tasks. The execution will first set the number of incoming dependencies for each vertex and it will start executing the vertices/tasks with no dependencies. When a task is completed it decrements a count on all its successor vertices (tasks). If other vertices have their count to zero then they will be scheduled for execution. Work stealing in this case happens only across ready to run tasks.

With the above support we anticipate users will be able to exploit parallelism decoupled from lower level details about the machine and thus allowing the runtime to perform the mapping to the machine. The concepts we introduced in this section are not new and they are currently employed in other libraries like Intel TBB, STAPL[7], Galois[10]. The novel part that we are focusing on in System G Native Store, is to employ these patterns to generalize some of the existing programming models like Pregel [9], Giraph [2] that target mainly fully parallel vertex centric computations disregarding for example the fact that you may need to process vertices in a particular order.

Vertices	Size	Properties	Load Time(s)
Person	100,000	8	0.45
Post	54,784,723	7	188.8
Forum	3,676,271	3	10.6
Place	5,130	3	0.01
Tag	12,144	3	0.03
Edges			
Person-knows-person	2,887,797	0	3.21
Person-likes-post	208,241,439	0	311
Post-hasCreator-person	54,784,723	0	54
Post-hasTag-tag	42,797,703	0	34
Forum-contains-post	54,784,723	0	52

Figure 8: Input dataset used for evaluation

6. PERFORMANCE EVALUATION

We evaluate in this section the performance of our graph library and runtime system using benchmarks that import a large corpus of data into the graph datastructure and subsequently performs a set of queries on the graph.

We use for our experimental evaluation a social network dataset generated using the "Linked Data Benchmark Council (LDBC)" graph generator. The dataset is generated apriori and stored in CSV files on the disk. The vertices and edges considered for the experiments in this section are described in Figure 8.

6.1 Graph construction

We imported the input dataset as described in Figure 8 using an Intel Haswell server with two processors, each processor 12 cores running at 2.7GHz and 256 GB of memory. The input CSV files are stored on an SSD disk. The graph is built using one thread, in memory only and is the multi-property graph as described in Section 4.3. We include in

Figure 8 in the fourth column the time it took to read individual input files and add corresponding vertices and edges in memory. In addition to the corresponding add vertex and edge methods the execution time includes reading from file, parsing input, add related properties. It also includes the execution time to add entries into an index from an external vertex identifier specified in the input files to the internal vertex identifier used by our graph library. In general the ingestion time depends on the number of properties considered. For example when adding the post vertices we achieve a rate of 290,000 vertices per second.

6.2 Graph queries

In this section we evaluate three different queries performed on the dataset we considered. LDBC project provides a set of seven sample queries and in this work we evaluated query number two, four and six[1], denoted by Query 1, 2, and 3 in this paper, respectively. Query 1 finds the newest 20 posts among one person' friends; Query 2 finds the top 10 most popular topics/tags (by the number of comments and posts) that ones friends have been talking about in the last x hours; and Query 3 finds 10 most popular tags by people that are among your friends and friend-of-friends, which appear in posts where tag x is also mentioned. The queries are implemented exactly as specified on the LDBC site, retrieving all the required fields. Essentially, all queries are localized searches through the graph datastructure starting from a particular vertex and subject to various filtering conditions. One can notice the importance of having labels for edges and more importantly having an efficient storage that will allow traversals of only edges and vertices of a particular label.

In Figure 9 we illustrate the distribution of 1000 query (Query 1) times where the starting vertex was randomly chosen. The bars depict how many queries had the average execution time shown on X axis, while the line and the right Y axis show the average number of edges traversed per query for a particular bucket. From the figure we notice first that for this types of graphs most queries have a relatively low number of edges traversed (1000 for the first bar). However for a small number of queries the number of edges traversed is close to 70k. Secondly, as expected the execution time increases with the number of edges traversed by query but for all queries considered the time was under 25 microseconds. The throughput of the queries is shown in Figure 10, where the metric on the y-axis is the number of traversed edges per second (TEPS). The figure clearly shows that System G Native Store scales well for various number of threads and all the queries, and the throughput is excellent, reaching 160 million edges per second.

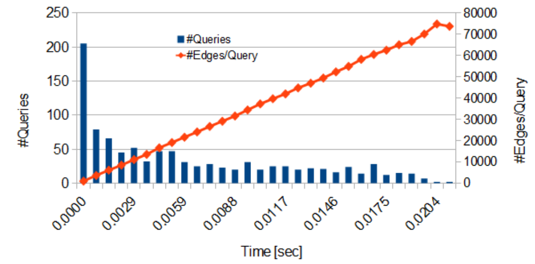

Figure 9: Distribution of Query 1 Times

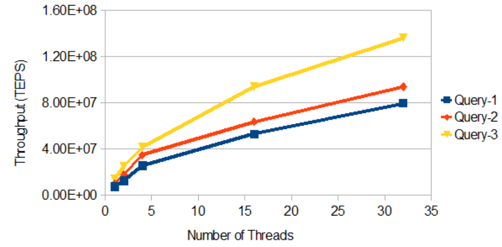

Figure 10: Throughput of Graph Queries

7. CONCLUSION AND FUTURE WORK

This paper presents a high performance graph library in IBM System G Native Store, a whole spectrum solution for graph processing. This runtime library is not only easy to use due to its intuitive APIs, but is also highly efficient for various graph analytics. It preserves graph data locality and supports multiproperty graphs, persistent graphs, concurrent and distributed graphs. Preliminary experiments show highly efficient graph processing performance using a social network dataset with 500 million edges.

[1]https://github.com/ldbc/ldbc_socialnet_bm_neo4j/wiki/Queries

8. REFERENCES

[1] SPARQL. http://www.w3.org/TR/rdf-sparql-query.

[2] Apache giraph. https://giraph.apache.org/, 2014.

[3] Tinkerpop. http://www.tinkerpop.com/, 2014.

[4] Titan distributed graph database. http://thinkaurelius.github.io/titan/, 2014.

[5] M. Canim and Y. Chang. System G data store: Big, rich graph data analytics in the cloud. In *IEEE International Conference on Cloud Engineering*, 2013.

[6] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing*, 2005.

[7] Harshvardhan, A. Fidel, N. Amato, and L. Rauchwerger. The stapl parallel graph library. In *Languages and Compilers for Parallel Computing*, 2013.

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[10] K. Pingali. High-speed graph analytics with the galois system. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 41–42, 2014.

[11] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013.

[12] J. Siek, A. Lumsdaine, and L.-Q. Lee. Boost graph library, http://www.boost.org/libs/graph/doc/index.html. 2001.