Explore Efficient Data Organization for Large Scale Graph Analytics and Storage

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

***Abstract*—Many Big Data analytics essentially explore the relationship among interconnected entities, which are naturally represented as graphs. However, due to the irregular data access patterns in the graph computations, it remains a fundamental challenge to deliver highly efficient solutions for large scale graph analytics. Such inefficiency restricts the utilization of many graph algorithms in Big Data scenarios. To address the performance issues in large scale graph analytics, we develop a graph processing system called System G, which explores efficient graph data organization for parallel computing architectures. We discuss various graph data organizations and their impact on data locality during graph traversals, which results in various cache performance behavior on processor side. In addition, we analyze data parallelism from architecture's perspective and experimentally show the efficiency for System G based graph analytics. We present experimental results for commodity multicore clusters and IBM PERCS supercomputers to illustrate the performance of System G for large scale graph analytics.**

***Keywords*-Graph Processing, Parallel Computing, Scalability**

I. Introduction

Large scale graph analytics and storage systems have been drawing increasing attention 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 graphs.

`Challenges`: Similar to many Big Data problems, a major challenge for large scale graph processing is *performance*, that is, handling a vast amount of linked data within a reasonable amount of time (sometimes in real time). Due to the *large volume* of data and highly *irregular* data access patterns, performance is particularly challenging for graph processing systems [6]. Such irregularity makes some MapReduce-based Big Data tools less suitable than tools built specifically to handle graph problems. [12][14][3].

- *Scalability* is challenging for all Big Data problems, including Big Graphs. Most existing work on graph scalability, especially solutions based on MapReduce, focus on *scaling out* rather than *scaling up*, which can result in poor sustained performance on every single machine. By scaling up, we mean accelerating graph analytics using large shared memory nodes and leveraging the multicores and optimizing cache performance, etc. GraphLab has demonstrated superior performance compared to scale-out Hadoop-based solutions using far more computers. [10].
- *Architecture-awareness* is the approach to exploit the platform capability efficiently. It is known that the sustained performance of most applications is no more than 20% of the peak performance of the platforms [17]. To design a large scale graph processing system that maximizes performance, one must deeply understand the primitives in graph processing, such as graph traversal and update, and (re-)design the algorithms for them based on the underlying architecture.
- *Systematic-optimizations* are critical for a whole spectrum solution of large scale graph processing systems. Many existing solutions focus on a specific tier, which may result in inefficiency for building graph analytics across multiple tiers. For example, GraphLab has limited capability to handle disk IO, which possibly results in high overhead in graph persistency. Neo4j organizes graph data amenable to disk IO, but may degrade CPU cache performance in graph analytics due to the scattered adjacent edges for a vertex.

`Contributions`: System G is a whole spectrum solution for large scale graph processing, including graph storage, runtime, analytics and visualization. In this paper, we focus on the performance aspect of the System G runtime. The main contributions include:

1) *Scalability*: We address the parallel-amenable design of graph data representation in the concurrent graph runtime library and distributed library in System G. we experimentally show the superior scalability of several graph analytics, such as Breadth first search (BFS), single source shortest path (SSSP), and loopy Bayesian inference, implemented on multicore processors and POWER7+ clusters, using 1 to 4096 threads/processes.
2) *Architecture-awareness*: We proposed cache-friendly design for large scale graph processing on commodity multicore processors and clusters. We use the graph traversal as an example to illustrate efficient graph data layout and an efficient dynamic memory management based on lock-free extensible circular queues. We experimentally discuss the impact of emerging architectures, such as remote directed memory access (RDMA) and solid state

drive (SSD) based active storage, on the performance of graph processing systems.

3) *Systematic-optimization*: We present a flexible graph APIs based on STL-like C++ template programming, which allows any type of user-defined vertex and edges. Any graph analytics built on top of the APIs can take advantage of the efficient runtime libraries. It can also use the provided graph storage implementation for persistence. We demonstrate the performance of graph analytics on both small and large scale graphs and the preloaded execution time can be in near real-time even for a graph with 10 billion vertices across 128 machines.

This paper is organized as follows: We address the background and related work in Section II. Following a brief description of System G in Section III, we discuss our graph data organization and its performance tradeoffs in Section IV. We then show various experimental studies in Section VIII and then conclude this paper in Section IX.

II. Background and Related Work

We focus on graph databases which provides comprehensive graph data management as well as graph runtime support. This is in contrast to a set of packages called graph computing framework, such as Pregel [12], Giraph [2], GraphLab [11], which provides limited support on persistency or simply relies on non-graph databases.

Neo4J [14] provides a disk-based, pointer-chasing graph storage model. It stores graph vertices and edges in a de-normalized, 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. SAP HANA [15] offers a graph engine on top of the column store shared with another relational (aka., SQL) engine. Compared with other graph stores built on relational storage and relational query engine, it has a low-level graph abstraction layer and graph-specific query engine. Certain queries such as graph traversal has been proved to be much faster than using the SQL engine. Trinity [16] is a graph store on top of distributed memory cloud, and it spreads graph data into the memory of multiple servers using consistent hashing. Compared to many disk-oriented solutions it achieves much better performance in both ad hoc query and batch analytics.

There are a lot of RDF stores that persist graphs in triple store and implement SPARQL interface on top of that [5]. However it is known that RDF and SPARQL is not good at answering traversal type of graph queries.

Some recent literature such as GBase [7] discusses graph storage based on block compression, uses matrix-vector operation to unify all graph operations, and implements a set of graph algorithms using Hadoop. The authors of [18] proposed a native graph storage called Trinity for RDF graphs and shows improved performance over relational storages. GraphChi [8] proposed a novel method called Parallel Sliding Windows (PSW) to process large graphs from disk, while requiring minimal random disk access. Lee and Liu [9] present a semantic hash partitioning approach that use the semantics embedded in RDF triples to improve the locality of graph partitioning and query.

TABLE I
Comparison with popular graph stores

	System G	**Neo4j**	**Titan**
Front-end	Graph, schemaless	Graph, schemaless	Graph, schema
Back-end	Graph files	Graph files	Non-graph
Query	Gremlin, gShell	REST, Cypher	REST, Gremlin
Language	C++, Java	Java	Java
Integrity	ACID	ACID	Eventual consistency
Compression	No	No	Yes
Performance	Superior	Good	Poor

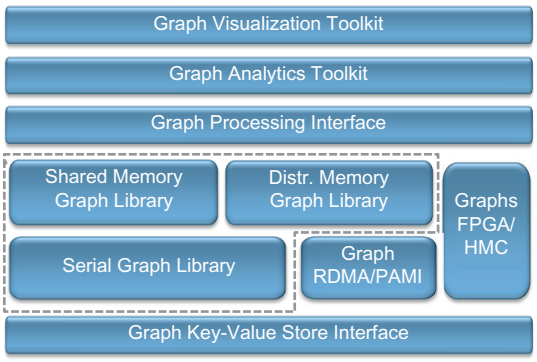


Fig. 1. A simplified overview of System G Architecture

A summary of the comparison between System G graph data-store and related work is shown in Table I.

III. System G Overview

System G is a comprehensive graph processing system for IBM Big Data, where *G* stands for graphs – large or small, static or dynamic, topological or semantic, properties or probabilistic. The system consists of four layers, which are the graph visualization toolkits, the graph analytics toolkits, the graph APIs and runtime libraries, and the graph key-value store. In this paper, we focus on the *graph runtime libraries*, as shown in the dashed polygon in Figure 1, which powers the entire system for large graph analytics. In particular, it provides efficient organization for high performance graph data access. The graph runtime libraries include:

`Serial graph library`: The System G graph runtime libraries include a serial graph library that implements the fundamental data structures for graphs, a vertex-centric graph representation with vertex/edge properties. Details on the graph data organization is discussed in Section IV.

`Shared memory graph library`: This library is built on top of the serial graph library to support concurrent graph operations on multithread computing platforms, including the servers equipped with multisocket processors, multicore processors, and manycore processors, such as IBM POWER, AMD Opteron, Intel Xeon/Phi, and Oracle/Sun

UltraSPARC. This library optimizes processor cache performance that is usually poor in graph processing due to irregular data access patterns.

`Distributed memory graph library`: This library handles graphs beyond the capacity of a single machine. To address the challenges in distributed graph access, we enabled graph communication layer using techniques such as the Remote Direct Memory Access (RDMA and the IBM Parallel Active Message Interface (PAMI) [13]. RDMA offers low latency one-side data access, which allows remote data retrieval without interrupting the remote processors; while PAMI has open-source implementations on various commodity computing platforms, which utilizes novel techniques to reduce communication overhead and support lock-less communication primitives.

`Graph processing interface`: All the above libraries are uniformly integrated under a STL-like C++ template-based API. Due to the flexibility of the template approach, it allows generic programming that can incorporate any user-defined types for vertex and edge properties.

`Graph key-value store interface`: The graph data is persisted on disk and can be accessed using this interface.

To simplify users' effort in writing parallel algorithms, the runtime libraries support a `task based model for parallelism`. The users are provided with high level constructs including:

- `for_each(graph* g, functor f)` - Invoke the functor `f` on each vertex of the graph in parallel.
- `schedule_task_graph(directed_acyclic _graph* g, functor f)` - Invoke the functor `f` on each vertex of the graph, possibly in parallel, making sure the execution respects the dependencies of the input directed acyclic graph (DAG). More details are included in Section VIII-D.
- `reduce(graph* g, functor f)` - Invoke the functor `f` on each vertex. The functor will return a value that is aggregated across all vertices and the result is returned to the user.

Existing computation patterns, like MapReduce, Pregel, can be expressed within our framework. Additionally, the task graph options allow users to express patterns that can't be achieved just by using MapReduce or Pregel.

IV. Graph Data Organization

A graph consists of a collection of vertices and edges. We use the notation $G(V, E)$ to denote a graph, where V represents the set of vertices and E represents the edges. Within our framework, we use a variant adjacency list to store the edges inclined to a vertex. Subsequently, the graph stores a set of vertices and each vertex individually maintains the list of its neighboring edges or its adjacency.

Our core data structure models a single property graph. In this model the graph stores a property buffer for each vertex and edge, regardless the semantic and internal structure of the property. Using the generic programming approach available in C++, a graph data structure thus is available to the user with the signature shown in Figure 2.

```
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;
```

Fig. 2. System G Graph Interface

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. For example, a user can declare a directed graph where each vertex has an integer property, and each edge has a double property as in Figure 2, Line 17. Another example is shown in line 19 and 20 where the vertex property is a user defined class and the graph is an undirected graph.

When deciding a particular graph representation, advanced users can choose the data structure for its vertices and edges. This can be either `list, vector or map based` with various memory versus execution time trade offs for access and modifying operations. By default we store vertices in a list, and for each vertex we store its edges as a list as well. The list provides the important property that element references are not invalidated when elements are added or deleted. This is a necessary requirement for our generic graphs but other specializations, like read only graph can use arrays to store vertices and edges for example.

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.

We briefly describe the main interface of the System G graph classes as followed. First, the function `add_vertex` shown in Figure 2, Line 10 creates a new vertex in the graph and returns the vertex descriptor associated with it. The vertex descriptor 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 and etc. The function `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. Having access to the iterator one can access all its properties including its adjacency. The interface includes a larger set of methods for deleting vertices, edges and more custom functionality that can't be all described here due to the lack of space.

A. Multi property graph

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

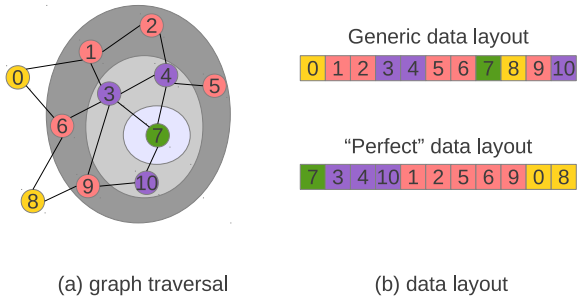
B. Concurrent and Distributed Graph

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.

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 that 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.

When running in a distributed environment often each node of the machine will add vertices locally but edges can be added from any node to any vertex thus we use RPC to achieve this. One of the main challenges with the approach we described is finding the location where a particular vertex lives. To achieve this we use a distributed hash table that stores the mapping from a particular vertex identifier to the location where the vertex resides. The distributed hash table uses a fixed formula to map from an identifier to the node of the machine where the value is stored. Thus accessing a remote vertex often implies two remote messages. A first one to lookup in the hash table for the location of the vertex and a second one to access the remote vertex on the location returned in the first step.



(a) graph traversal (b) data layout

Fig. 3. Example of the impact of different data layouts

C. Impact of Graph Layout on Cache Performance

Large scale graph analytics usually requires a distributed environment for parallel computing. However, when it comes to a single chunk of the workload running on one processor, cache performance becomes a critical issue because of the high memory intensity of graph analytic applications. On the other hand, graph analytic operations are considered to have poor cache performance, due to their irregular access behaviours. Their memory accesses follow a pointer-chasing pattern, which usually lacks data locality. Because of that, optimizing an individual instance of graph analytics is known to be difficult.

A proper data layout in memory can still significantly affect the cache performance of graph operations. An example is shown in Figure 3. In this example, a breadth-first-search (BFS) traverses the graph starting from vertex 7. A generic graph data layout would be organizing data by vertex number. However, it is not the optimal layout and will lead to extremely low cache hit rate. In the example of Figure 3, the "perfect" data layout should be an array of vertices following the pattern of access order. If the data size of each vertex is 16 bytes, the cache hit rate would be roughly 75% in regular caches with 64 bytes cache blocks. Based on above observations, graph data reorganization techniques were proposed. In those techniques, graph data is reorganized to match the exact access pattern of each query. Nevertheless, data access pattern of graph operations largely relies on specific dataset and query. Even different queries of the same operation can lead to significant variance in access patterns. Meanwhile, it is infeasible to reorganize data layout for each query in large scale graphs. The overhead of copying and moving large amount of data is much larger than the benefit of cache performance gains. Therefore, instead of fulfilling requirements of individual queries, the data layout of large scale graph should be a generic solution, which strikes a balance between the requirements of different graph analytic operations.

Besides cache performance itself, graph data layout also needs to be flexible enough to support timely graph updates, which may happen frequently for certain graph applications. However, flexible data layout usually leads to a data structure

with more pointer-chasing and therefore less cache-friendly. For instance, linked-list shows more flexibility than an array, but also less cache friendly. Therefore, an efficient graph data layout should be generic, cache-friendly, and flexible. To achieve all these three major goals, our proposed system, instead of focusing on tuning data structures for individual instances, leave it open for users. Users can make choices between different predefined data layouts of vertices and edges. Even user-defined data structures are also supported. In this way, users can gain largest performance benefit as well as flexibility.

V. Efficient Graph Traversal

A. Graph Traversal as a Primitive

A *graph traversal* visits all the (reachable) vertices in a graph or a subgraph, updating and/or checking their values along the way. This is a representative operation in graph processing, where the irregular data access patterns distinguishes graph traversal from other computations. Graph traversal serves as a fundamental operator in many graph analytics. Some well known graph operations based on traversal include the shortest path, copying collection, bipartite test, mesh numbering, maximum network flow, etc.

Therefore, a high performance graph processing system must deliver highly efficient graph traversal primitives. Designing such a primitive requires deep understanding of the platform architectures. System G runtime provides various solutions for the graph traversal primitive. Because the System G serial graph traversal has already been addressed in Section IV-C, we focus on concurrent and distributed solutions here.

B. Compact Graph Representation

For certain large graphs where real time solutions are required for traversals, a complex generic graph representation, may not be feasible due memory constraints. Compact representation of graph not only improve the efficiency of memory/storage usage, but also increases data locality as well as the cache performance. There are various ways to represent a graph. Well known formats include adjacency matrix and adjacency list. The former is straightforward to incorporate with matrix-based optimizations. For such an approach, the high performance solutions are relatively mature. However, this representation may not be friendly to large scale graphs which are usually sparse. The latter, i.e. adjacency list, scatters graph data in memory and therefore causes overheads due to highly irregular memory accesses.

We illustrate a highly compact graph representation in Figure 4. This is called the compressed vertex format (CVS), adapted from the compressed row format (CRS) widely used in sparse matrix processing [1]. The CVS uses a few long arrays to represent the data. Despite of the high compactness, the CVS format has an intrinsic issue to handle dynamic graph. For instance, if a new edge is inserted into the graph, it may result in significant data moving. An alternate solution is to partition the CVS format into segments (see Figure 4, where each chunk is still a compact CVS subgraph and the

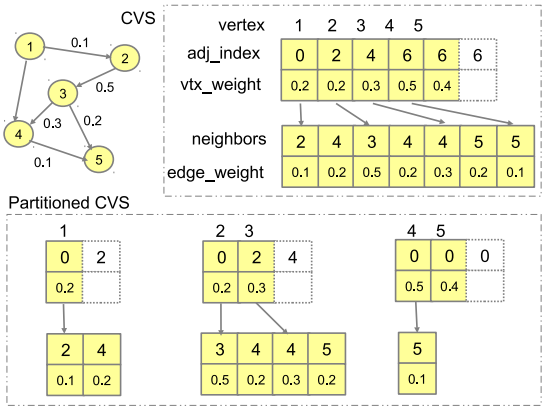


Fig. 4. Various Graph Representations

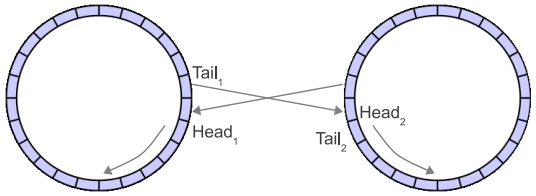


Fig. 5. Extendable Circular Queues

graph updates would only affect the corresponding subgraph. It is worth noting that when the partitioned CVS is in finest granularity, i.e., each vertex is separated, it degenerated to the traditional adjacency list.

C. Traversal with Extendable Queues

The *queue* is a key data structure used by graph traversal. For example, BFS is essentially to fetch (dequeue) a vertex from a queue, initialized with a queue with the root only, and then store (enqueue) its unvisited neighbors to the queue. Similarly, Dijkstra SSSP fetches a vertex from a queue and updates its neighbors' distances to the source; those with reduced distances are then enqueued. Some variant form of SSSP utilizes priority queue (heap) to improve the performance. Both the ordinary queue and the priority queue can be implemented based on arrays. We discuss the ordinary queue only in this section.

We use an extendable circular queue for delivering high performance for large graph analytics. The data structure for the queue is shown in Figure 5, which consists of one or multiple circular queues, each having a head and a tail. An element is dequeued from the head and enqueued to the tail, denoted by h_i and t_i. A larger circular queue can be built by merging two queues. By merging q_i and q_j, we just need to alter two pointers shown in Figure 5. Compared to the deque implementation based on double linked list, the proposed data organization has improved compactness and data locality; compared to the array based implementation, the proposed one

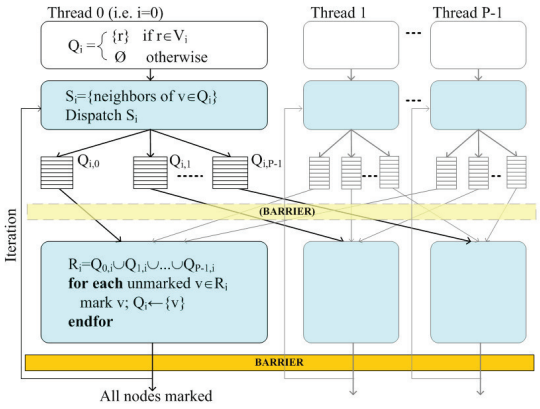


Fig. 6. Distributed Graph Traversal

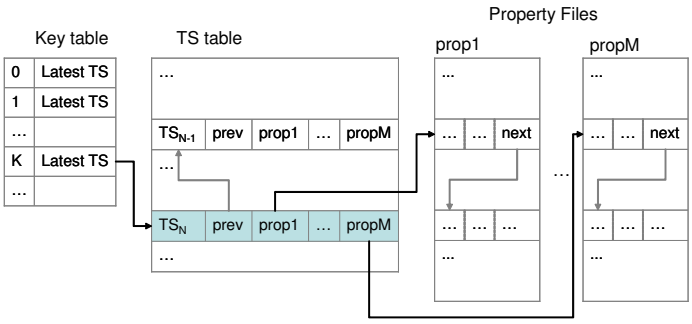


Fig. 7. The Structure of Native Store

allows one to easily extend the queue without moving any data.

Using such efficient data organization, we are able to perform BFS or SSSP on graphs with billions of vertices within a few seconds on a single machine.

D. Concurrent and Distributed Traversal

Concurrently accessing the queue during traversal must be synchronized in order to avoid conflicting updates to the graph data. Conventionally, this is done by mutex locks, such as the spin-lock. We observed that the overhead of mutex locks is high for large scale graph processing. A more efficient solution is to use the lock-free queues, which utilize hardware level atomic operations, such as Fetch&Add (FAA) and Compare&Swap (CSW) to replace locks. Such lock-free techniques can be directly applied to our circular queues.

For distributed traversal, the vertices in the queue should be distributed among processors. We illustrate a distributed graph traversal in Figure 6. The execution of each consists of alternating local computation steps and global barriers. Given P processors, each processor p_i partitions its queue into P parts, where $Q_{i,j}$ will be processed by p_j. Note that the communication overhead increases as the number of processors increases. To achieve scalability, each processor should have sufficient amount of local workload; otherwise, there is no need for parallelization.

VI. Persistency Challenges and Solutions

We save the graph data in a set of disk files to satisfy the persistence requirement. We provide a graph tailored key-value store interface as shown in Figure 1. Considering persistence and performance issues, we implemented the data store in C++ language with an edit log.

A. Native Store Structure

We save the vertices and edges of a graph separately in two set of native stores; while each native store is saved in several disk files. The fundamental structure is shown in Figure 7. The Key table, which is saved in a disk file, converts an internal sequential ID to an address inside the second table called Time Stamp table (TS), which is also saved in a disk file. As the size of each entity in the Key table is fixed, the internal ID can be used to compute the address of each entity directly by simply multiplying the ID with the entity size. To avoid gaps in the table due to vertex/edge deletion, the internal ID will be reused after it has been erased.

The time stamp (TS) table is used to implement the multi-version feature of the native store. We save TS table in a file, where we link the history of a vertex/edge together using a single linked list; the pointers in the Key table point to the first item of the list, which saves the latest version of the property. The remaining part of the table saves the pointers to the property values saved in the property files. Through this table, we are able to access the properties of an internal ID at a specific time stamp or its modification history.

For a native store, the kind of the properties saved and thus the number of property files can be configured as parameters of the TS table. For example, we can configure two properties for a vertex store, which are its adjacent edges and the vertex's name. Thus, each entry in the vertex TS saves the locations of the two properties.

B. Challenges and Solutions

While Figure 7 demonstrates the fundamental structure of the native store, we modify this structure as we face the challenges from both performance and persistence perspectives.

Since the native store is saved as disk files and disk IO is expensive, we pay much attention to reduce the IO number and change IO patterns from random access to sequential access. For a normal property retrieval operation for the native store, we need to have 1) one read of the Key table to get the address of the latest version entity in the TS table; 2) one read of the TS table to get the address of the property value inside the property file; 3) one or more reads of the property file depending on the size of the property value. In total, we have at least 3 read operations. To reduce the number of the read operations, we move the property pointers of the latest version from the TS table to the Key table. Thus, we eliminate one read access to the TS table when the latest properties are retrieved, which accounts for most of the data queries. One step further when there is enough memory, we can configure to have the Key table cached in memory and thus eliminate both read access to the Key and TS tables.

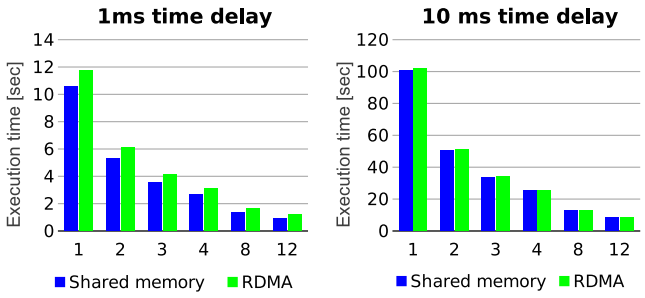


Fig. 8. Latency comparison between RDMA and local memory access

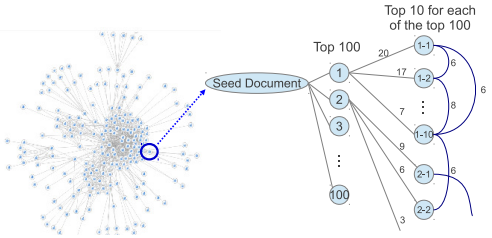


Fig. 9. A balanced subgraph query for visualizing recommendations

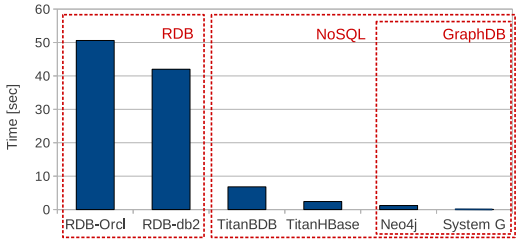


Fig. 10. Performance of graph traversal in various databases

We also implement an edit log, which is used to log the modifications to the data store. For an insert operation to the data store without using the edit log, we need at least 3 random write operations to insert to the Key table, the TS table and the property file. By using the edit log, we shall not only reduce the operations to one write but also the write is always at the end of the log file. Periodically or triggered by events, we shall sequentially read the edit log, merge the operations of the same key, and convert the content to the native store.

VII. Utilizing Emerging Architecture Technologies for Graph

Emerging architectures has deep impact on the efficiency of graph processing systems. We experimentally show the impact of some of these novel architecture trends on System G performance.

The remote directed memory access (RDMA) provides low latency remote data access. Due to the irregularity in graph computing patterns, data access latency can dominate graph processing time for distributed memory graphs. RDMA is a promising technique for distributed graph computing. We conducted experiments on a cluster with InfiniBand (IB) by using a graph traversal where a local computation was incurred when visiting a new vertex. Figure 8 shows that, when the latency of the local computation is about 10 ms, or more, the distributed implementation is as efficient as that on shared memory platforms.

Some other relevant architectural innovations that can benefit graph computing include SSD storage, hybrid many-core processors (GPU, Intel Phi, etc.) and FPGAs but they are not exploited by the current runtime release.

VIII. Experiments

A. Impact of Data Storage

We experimentally show the impact of data storage on the performance of graph analytics using a straightforward graph query for visualizing recommendation. We utilized a real data set from IBM, which consists of about 72,300 user vertices, 82,100 document vertices, and over 1,740,000 edges. Given a vertex of a document d, the graph query creates a subgraph consists of top 100 most relevant documents to d, and we built an edge between the found document and d; then, for each found document we find the top 10 documents again, as shown in Figure 9.

We implemented the query on graphs that were stored in various storages, including two relational database (Oracle and DB2), a K/V store (Berkeley DB) with Titan, a HBase with Titan, a graph store (Neo4j) and the proposed System G. Even thought the implemented algorithm is the same, the performance in terms of execution time varies significantly. The two relationship database (RDB) are much slower, possibly due to the overhead in table joins. The graph stores (Neo4j and System G) shows superior performance (1.2 and 0.7 seconds, respectively), since the data is organized both in memory and on disk as a graph, resulting in efficient graph traversal and updates. The striking contrast justifies the importance of graph data store for large scale graph analytics. Neo4j is based on JVM, so it is hard to deep optimize on particular platforms. System G shows improved performance compared to Neo4j implementation in this experiments, because of our efforts on optimizing the cache performance.

In Figure 11, we compare the throughput (number of edges traversed per second, TEPS) of graph travesal with respect to two baselines (Neo4j, Titan with BDB), using graphs of various sizes. System G significantly outperforms the baselines, especially for Titan. This is because the back end of Titan is not a graph store, which can not be optimized towards graph computing characteristics.

B. Large Scale Graph Traversal

We conducted experiments on two machines to evaluate the graph traversal performance of System G. The two platforms are:

- `sg20` is a server with Intel Xeon E7-4830 running at 2.13 GHz and 256 GB DDR3 memory. There are 64 vir-

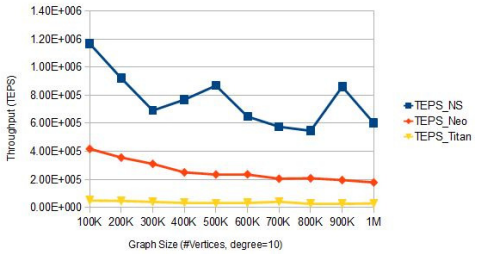


Fig. 11. Graph traversal throughput comparison

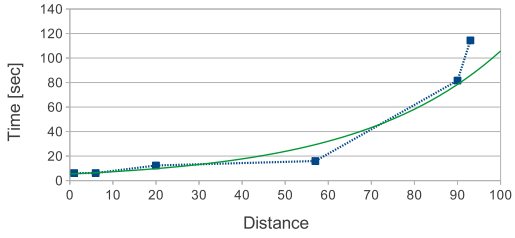


Fig. 12. Execution time on finding shortest paths on a graph with 1-billion vertices

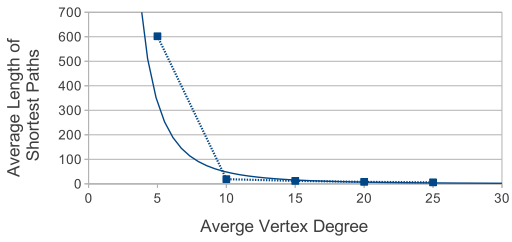


Fig. 13. Impact of average node degrees on the average lengths of shortest path.

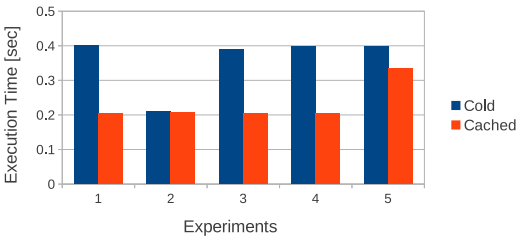


Fig. 14. Impact of cached traversal data on the execution time on finding shortest paths.

tual processors with hyperthread enabled. The operating system is Centos Linux.

- PERCH is a POWER7+ cluster with 128 compute node, each with 32 virtual processes with SMT enabled. Each node has about 90 GB memory available. The total number of processes is 4096 and the total memory capacity is about 1.5 TB. Unified Parallel C (UPC) is installed.

On sg20, we loaded a sparse undirected graph with 1 billion vertices. The number of edges is about 10 billion. We randomly select a set of vertex pairs, say (x, y) and find the single source shortest path (SSSP) between x and y. We found the shortest path by traversing the graph from both x and y, which is similar to Dijkstra algorithm. During the process, once a vertex is visited by both traversals, we check if a shortest path is found. We sort the vertex pairs according to the found shortest distance and show their corresponding execution time in Figure 12. It clearly shows that, if the shortest path is short (say, within 10 hops), System G finds it in a few seconds.

In many analytics, if the shortest path between two vertices is too long, it is not necessary to find it precisely. We experimentally show the impact of the average distance of shortest paths on the execution time. By average distance, we mean the average of the distances between vertex pairs randomly selected. We randomly generated graphs with 1 million nodes with various graph densities. The results are shown in Figure 13. The figure shows that the average shortest path distance decreases significantly as the graph density increases. Thus, it is reasonable to limit the length when finding shortest paths in a large scale graph processing system.

The traversal in System G can be cached. We show the improvement due to the cached traversals in Figure 14. Basically, we sorted the randomly selected vertex pairs. We intentionally choose those with the same source or target vertex. Thus, once a traversal starts, we check if it is cached. If it is, we do not perform the traversal but instead read the cached results. It shows that the cache accelerate to execution around 2x. For the second experiment in Figure 14, the path is so short that the traversal from the target vertex was not performed.

We performed SSSP in a graph with 10 billions nodes and 200 billion edges on PERCH. The implementation is based on UPC using the partitioned global address space (PGAS) model. Powered by the 4096 processes, the execution time is promising despite of the extremely large scale of the graph. The results are shown in Table II.

C. Cache Performance of Graph Traversal

To better understand the behaviour of graph analytics, we performed multiple experiments to analyze its cache performance. McSimA+, an event-driven timing simulator [4], was used to analyze the impact of different cache structures and data layouts. McSimA+ is a many-core simulation infrastructure where out-of-order cores, caches, directories, on-chip networks, and memory channels are modeled. The simulator was configured to have 64B cache line size for both L1 and L2 caches. L1 cache size varies from 8KB to 128KB, while

TABLE II
EXECUTION TIME (SEC) IN FINDING SHORTEST PATHS ON PERCH

Path Distance	2	4	8
1-Billion Vertices Graph	x	0.421	x
10-Billion Vertices Graph	0.413	0.421	x

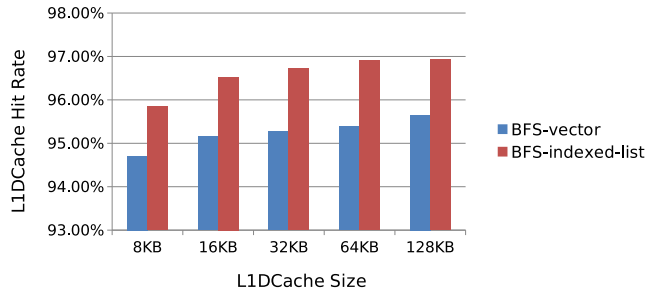


Fig. 15. L1 cache hit rate with different cache sizes

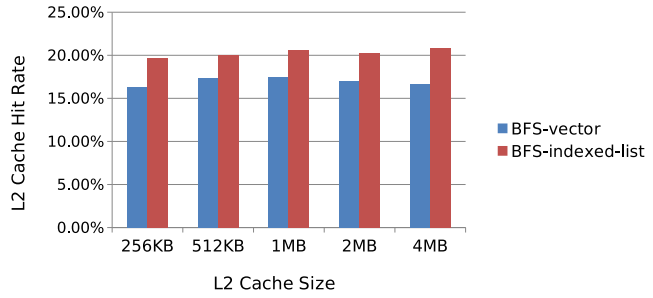


Fig. 16. L2 cache hit rate with different cache sizes

L2 cache has a size between 256KB and 4MB. Although core simulation is usually considered as an inaccurate method to measure performance, its cache hierarchy model is sophisticated enough to achieve accurate hit rate result. Thus, we performed our analysis via simulation experiments.

In the experiments, we chose BFS graph traversal, the most representative graph operation, as the benchmark. Graph initialization phase was skipped during the simulation, leaving only graph traversal simulated in details. Due to the limitation of simulation speed, we ran our experiments on a relatively smaller graph dataset with 5,000 edges. Considering the size of L1/L2 caches, the dataset size is large enough to illustrate the behavior of cache hierarchy. To further analyze the data layout's impact, we picked two different layouts, a vector-like layout, and a list-like layout. Both layouts are following a vertex-centric graph representation, in which a vertex is the basic unit of a graph. The vertex property and the outgoing edges stay within the same vertex structure. In the vector-like layout, the vertices are organized as a vector following a compact and sequential manner, while in the list-like layout, the vertices are chained together as a linked-list. In addition to that, the list-like layout also maintains an index by using an extra hash-map for fast vertex lookup.

The experiment results are shown in Figure 15 and Figure 16. From the results in Figure 15, we can see that although graph traversal lacks data locality, it still can achieve high hit rate in L1 data cache. Such behavior is caused by the data structures outside of the graph. Graph traversal has a simple algorithm and small code base. It is heavily accessing the program stack and the queue of vertex pointers. These key data structures both have small data size and good data locality. Thus, high hit rate in a small L1 cache was shown. We can also see that with larger L1 cache size, the hit rate increases slightly. In Figure 16 instead, the hit rate is low for all cache sizes. This is because memory accesses with high data locality are already filtered out by L1 cache. The accesses of L2 cache usually are coming from accesses with large data footprint or poor locality. In our experiments specifically, these accesses are generated by actual graph data traversal, which follows a pointer-chasing pattern with both large footprint and poor locality. Therefore, a low L2 cache hit rate is shown. In addition, because of the same reason, different L2 cache sizes show the similar hit rate without any benefit coming from larger cache sizes.

Comparing the cache performance results of different data layouts, Figure 15 and Figure 16 shows that a list-like layout has better hit rate than a vector-like layout in both L1 and L2 caches. By introducing more pointers to jump around, list-like data layout decrease cache performance in most cases. On the contrary, in our experiments, higher cache performance was observed. This is because of the special behaviour of graph traversal and the existence of the index. To walk through all vertices, graph traversal consists numerous pointer-chasing operations. Due to the lack of locality, neighbour data elements in the vector data layout will not be reused by later accesses. Thus, the compact layout of vector cannot bring benefits in cache performance. In contrast, the hash index in the list-like layout improves vertex lookup time and reduces unnecessary cache accesses as well. The reduction of poor locality accesses eventually helps both execution time and cache performance.

D. Scalability of Loopy Belief

In this section we analyze the performance of loopy Bayesian network inference. This computation is very interesting from a runtime point of view as it requires some advanced functionality in order to achieve good scalability. At the high level the computation uses as input a directed acyclic graph (DAG) and vertices can be processed in parallel as long as all their predecessors have been processed. The graph has one or more starting vertices or vertices with no dependencies.

To accommodate this pattern the runtime currently provides a DAG scheduler that automates all aspect related to thread and dependency management. The user often only needs to specify the work function that will be invoked by the scheduler on each vertex.

Work stealing optimization: In the loopy inference case the computation is proportional with the number of predecessors and successor nodes, thus leading to a variable amount of work per vertex. Consequently a static partition of the input DAG vertices and their mapping to individual threads is very difficult to achieve. For this reason our scheduler currently allocate the initial starting in a round robin fashion across all worker threads, while the additional vertices that are enabled as computation proceeds are spread across threads using work stealing. This very popular technique is expected to improve

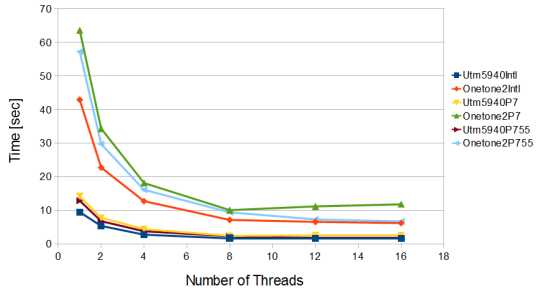


Fig. 17. Loopy Inference scaling for two different inputs and three different platforms (P7, P7IH and Intel)

the performance of all parallel computations where the amount of work per data item is variable.

Nested Parallelism optimization: A second optimization that we need to deploy in our runtime to further improve the performance of loopy like computations is nested parallelism. While work stealing allows threads that finish faster to help with computations of other threads, it sometimes happens that one particular computation in a thread is as long or longer then all the other computations in the other threads. In this case, if possible, the user will express the work on a vertex as a parallel computation which in turns get decomposed in a number of smaller tasks that other threads can help with the execution.

In Figure 17 we show the performance of the loopy inference when all the above described optimizations are used. We show the performance for two input graphs and three different platforms. An Intel Sandy Bridge with 2 CPUs, 4cores/CPU and up to 3.78GHZ with turbo boost, a Power 7 server with one CPU, 8 cores/CPU and running at 3.1GHz and a node of the PERCH server with 4 Power7 CPUs, 8 cores/CPU at 3.1GHz. The UTM5840 input graph has 5940 vertices and 57295 edges and we observe good scaling from 1 to 4 threads. Adding more threads helps minimally as the amount of work per thread decreases to the point where runtime and scheduling overhead are becoming an important component of the whole execution time. The second input graph (OnetoOne2) has 36057 vertices and 206205 edges. In this case we observe good scaling up to 8 threads on all platforms. On PERCH the computation scales up to 16 threads, while Intel and single CPU Power 7 stop scaling after 12 and 8 threads respectively.

The benefits we enable through the parallel execution mode supported in our framework allows users to either run bigger problems or run a given problem faster and either of this can be beneficial to users in various application domains.

IX. Conclusion and Future Work

In this paper, we proposed a large scale graph processing system called System G. This system focuses on the performance issues in processing large scale graphs on both commodity and high performance computing platforms. We particularly discussed the representation of large scale graph with properties on vertices and edges. We examined the impact of graph representation and computations on cache performance. We also inspected the traversal performance in graphs and the efficient data structure for both serial and parallel executions. Experimental results were provided to evaluate the proposed techniques towards large scale graphs. The results shows superior performance of System G for primitives in large scale graph analytics. These features distinguish System G from many Open Source based graph processing systems.

In the future, we plan to optimize the graph runtime libraries in System G for RDF graphs, where the vertices and adjacent edges of different types are stored separately. We would also like to further enhance the capability of System G towards highly concurrent graph accesses for supporting large scale graph analytics efficiently. Besides, the efforts on incorporating emerging architecture techniques will be strengthened to further improve the performance.

References

[1] Crs format. http://www.cise.ufl.edu/research/sparse/.
[2] Apache giraph. https://giraph.apache.org/, 2014.
[3] Titan distributed graph database. http://thinkaurelius.github.io/titan/, 2014.
[4] J. H. Ahn, S. Li, O. Seongil, and N. Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 74–85, April 2013.
[5] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *SIGMOD*, pages 121–132. ACM, 2013.
[6] G. Cong and D. A. Bader. Techniques for designing efficient parallel graph algorithms for smps and multicore processors. In *PDPTA*, pages 137–147, 2007.
[7] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD*, pages 1091–1099. ACM, 2011.
[8] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 8, pages 31–46, 2012.
[9] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. In *VLDB*, pages 1894–1905. VLDB Endowment, 2013.
[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, pages 1894–1905. VLDB Endowment, 2012.
[11] 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.
[12] 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.
[13] Redbooks, IBM. *PAMI Programming Guide*. 2011. http://publib.boulder.ibm.com/epubs/pdf/a2322730.pdf.
[14] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013.
[15] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The graph story of the sap hana database. In *BTW*, pages 403–420, 2013.
[16] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516. ACM, 2013.
[17] T. Sterling, P. Kogge, W. J. Dally, S. Scott, W. Gropp, D. Keyes, and P. Beckman. Multi-core for hpc: Breakthrough or breakdown? In *Supercomputing*, pages 1–12, 2006.
[18] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *VLDB*, pages 265–276. VLDB Endowment, 2013.