C-Graph: A Highly Efficient Concurrent Graph Reachability Query Framework

Li Zhou[*]
The Ohio State University
zholi@cse.ohio-state.edu

Ren Chen[*]
Huawei Research America
ren.chen@huawei.com

Yinglong Xia
Huawei Research America
yinglong.xia@huawei.com

Radu Teodorescu
The Ohio State University
teodores@cse.ohio-state.edu

ABSTRACT

Many big data analytics applications explore a set of related entities, which are naturally modeled as graph. However, graph processing is notorious for its performance challenges due to random data access patterns, especially for large data volumes. Solving these challenges is critical to the performance of industry-scale applications. In contrast to most prior works, which focus on accelerating a single graph processing task, in industrial practice we consider multiple graph processing tasks running concurrently, such as a group of queries issued simultaneously to the same graph. In this paper, we present an edge-set based graph traversal framework called C-Graph (i.e. Concurrent Graph), running on a distributed infrastructure, that achieves both high concurrency and efficiency for k-hop reachability queries. The proposed framework maintains global vertex states to facilitate graph traversals, and supports both synchronous and asynchronous communication. In this study, we decompose a set of graph processing tasks into local traversals and analyze their performance on C-Graph. More specifically, we optimize the organization of the physical edge-set and explore the shared subgraphs. We experimentally show that our proposed framework outperforms several baseline methods.

CCS CONCEPTS

• **Computing methodologies → Distributed algorithms**; **Concurrent algorithms**; *Massively parallel algorithms*;

KEYWORDS

Graph Processing, Concurrent Queries, K-Hop Reachability, Distributed System

ACM Reference Format:
Li Zhou, Ren Chen, Yinglong Xia, and Radu Teodorescu. 2018. C-Graph: A Highly Efficient Concurrent Graph Reachability Query Framework. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225136

1 INTRODUCTION

Graph processing has been widely adopted in big data analytics and plays an increasingly important role in knowledge graph and machine learning applications [11, 12, 39]. Many real-world scenarios such as social networks, web graphs, wireless network, etc., are naturally represented as large-scale graphs [8, 17]. Modeling applications as graphs provides an intuitive representation that allows exploration and extraction of valuable information from data. For example, in recommendation systems, information about neighbors is analyzed in order to predict the user's interests and improve click-through rate (CTR). High performance graph processing also benefits a wealth of important algorithms. For instance, mapping applications make extensive use of shortest path graph traversal algorithms for navigation. In order to effectively manage and process graphs, graph databases such as JanusGraph [4], Neo4j [29] and others have been developed. Graph processing frameworks are also commonly found as critical components in many big data computing platforms, such as Giraph in Hadoop, GraphX in Spark, Gelly in Flink, etc [2, 10, 30, 37].

One of the fundamental operations that a graph processing system must handle efficiently is the graph traversal. For example, the "reachability query" is essentially a graph traversal to search for a possible path between two given vertices in a graph. Graph queries are often associated with constrains such as a mandatory set of vertices and/or edges to visit, or a maximum number of hops to reach a destination. In weighted graphs, such as those used in modeling software-defined-networks (SDNs), a path query must be subject to some distance constraints in order to meet quality-of-service latency requirements [38].

Many real-world applications rely on *k-hop* [5], a variant of the classic reachability query problem. In *k-hop* the distance from a given node often indicates the level of influence. For example, in wireless, sensor or social networks the signal/influence of a node degrades with distance. The potential candidate of interest is often found within a small number of hops. Real-world networks are generally tightly connected, making *k-hop* query very relevant. According to the "six degrees of separation" principle, which claims that a maximum of six steps are needed to connect any two people, most of the network will be visited within a small number of hops. As a results, *k-hop* reachability often exists as an intermediate "operator" between low-level database and high-level algorithms [19].

[*] Contributed equally to this paper.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225136

Many higher-level analyses can be described and implemented in terms of *k-hop* queries, such as triangle counting which is equivalent to finding vertices that are within 1 and 2-hop neighbors of the same vertex. Therefore, a graph processing system's ability to handle *k-hop* access patterns predicts its performance on higher-level analyses.

Compared to many big data systems, graph processing generally faces significant performance challenges. One such challenge for graph traversals is poor data locality due to irregular data access patterns in many graph problems. As a result, graph processing is typically bound by a platform's I/O latency, rather than its compute throughput [9, 31]. In distributed systems the overheads of communication beyond machine boundaries, such as network latency, exacerbate I/O bottlenecks faced by graph processing systems.

Another challenge for most existing graph processing frameworks is to efficiently handle concurrent queries. These systems are often optimized to either improve performance or reduce I/O overhead, but are not capable of responding to concurrent queries. In enterprise applications, a system usually has to gracefully handle multiple queries at the same time. Also, since multi-user setups are common, several users can send out query requests simultaneously. Graph databases are often designed with concurrency in mind, but they generally have poor performance in graph analysis, especially in terms of handling large-scale graphs or high volumes of concurrent queries [31].

In this paper, we propose an efficient distributed concurrent query framework called *C-Graph*, short for Concurrent Graph processing system, to process concurrent local graph traversal tasks such as those in the *k-hop* reachability query. We take a high level view of the graph processing system design. While improving the efficiency of each processing unit, we consider both disk I/O and network I/O as elements of the storage bandwidth. *C-Graph* is designed as a traditional edge-centric sharding-based graph processing system. The main contributions of our framework can be summarized as follows:

- A simple range-based partition is adopted to reduce the overhead of complex partitioning scheme for large-scale graphs. Multi-mode, edge-set-based graph data structures optimized for sparsity and cache locality are used in each partition to achieve the best performance for different access patterns.
- The framework explores data locality between overlapped subgraphs, and utilize bitwise operations and shared global states for efficient graph traversals.
- In order to reduce the memory consumption of concurrent graph queries in a single instance, we utilize dynamic resource allocation during graph traversals. Instead of saving a value per vertex, we only store vertex values for those in the previous and current levels.
- Synchronous/asynchronous update models are supported for different types of graph applications, such as graph traversals and iterative computation (e.g. PageRank).
- Our system targets on the reduction of the average response times for concurrent graph queries on large-scale graphs with up to 100 billion edges in distributed environments.

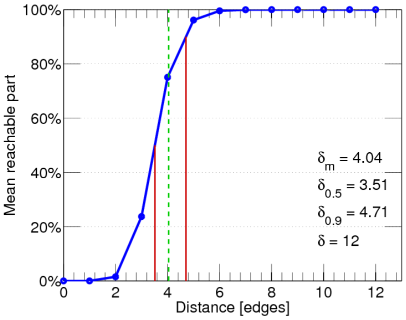


Figure 1: The hop plot for Slashdot Zoo graphs [15].

The rest of this paper is organized as follows: we first introduce the background of graph traversals and concurrent queries in section 2, then we discuss the main features and programming interface of our distributed query system in section 3. In section 4 we evaluate our system performance and scalability. Lastly, we discuss related work in section 5 and conclude in section 6.

2 BACKGROUND

A graph is denoted by $G = (V, E)$, where V is a set of vertices and E is a set of edges connecting the vertices; an edge $e = \{s, t, w\} \in E$ is a directed link from vertex s to t, with weight w for a weighted graph. Note that in graph database terminology the weight w can also be referred to as the property of edge e.

Graph Traversal and *k-hop* Reachability Query. Graph traversal is the process of visiting a graph by starting from a given source vertex (a.k.a. the root) and then following the adjacency edges in certain patterns to visit the reachable neighborhood iteratively. Examples of basic graph traversal methods include visiting a graph in breadth-first-search (BFS) and/or depth-first-search (DFS) manners. Most graph applications or queries are essentially performing computations on the vertex values and/or edge weights while traversing the graph structure. For example, the single-source-shortest-path (SSSP) algorithm finds the shortest paths from a given source vertex to other vertices in the graph by accumulating the shortest path weights on each vertex with respect to the root.

The *k-hop* reachability query [5] is essentially a local traversal in a graph, which starts from a given source vertex and visits vertices within k hops. It is a widely used building block in graph applications. In practice, the influence of a vertex usually decreases as the number of hops increases. Therefore, for most applications, potential candidates will be found within a small number of hops. In addition, real-world networks are often tightly connected. For example, Figure 1 shows the cumulative distribution of path lengths over all vertex pairs in the Slashdot Zoo network. In this network, the diameter (δ) equals 12. The 50-percentile effective diameter ($\delta_{0.5}$) equals 3.51, and 90-percentile effective diameter ($\delta_{0.9}$) equals 4.71. So most of the network will be visited with less than 5 hops, which is consistent with the six-degrees-of-separation theory in social networks.

The *k-hop* query is frequently employed as an intermediate "operator" between low-level databases and high-level algorithms [19]. Many higher-level functions such as triangle counting, which is

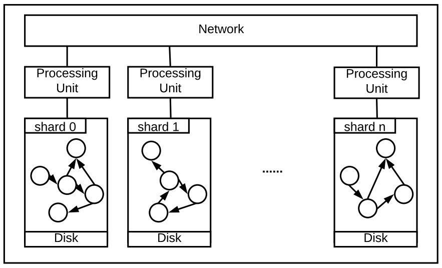


Figure 2: Overview of edge-centric sharding-based graph processing system design.

equivalent to finding vertices that are within 1 and 2-hop neighbors of the same vertex, can be described and implemented in terms of *k-hop* traversal. Breadth-first-search (BFS) is a special case of *k-hop*, where k → ∞. As a result, a graph database's ability to handle *k-hop* access patterns is a good predictor of its performance.

Concurrent Queries in Large-scale Graphs. The ability to handle concurrent queries is very important for industrial big data products. However, adding concurrency support in graph databases or graph processing systems is challenging. For example, graph databases like Titan [3], JanusGraph (based on Titan) [4] and Neo4j [22] are designed with multi-query/user in mind. However, their performance when executing concurrent graph queries is generally poor. In our experiments Titan took 10 seconds on average to complete 100 concurrent *3-hop* queries for a graph of 100 million edges. For some of the queries, the response time was as high as 100 seconds. Other graph databases like Neo4j are not distributed and cannot, as a result, support many real world graphs such as web-scale graphs partitioned over multiple machines.

High memory footprint is another challenge for large-scale graph processing. Concurrent graph queries, generally have high memory usage, which can significantly degrade the response times for all queries. As a result, most of the graph processing systems can't be easily changed to run concurrent queries. These systems are usually highly optimized for certain applications with high resources utilization, but system failures may be triggered when running concurrent queries due to memory exhaustion.

3 SYSTEM DESIGN

In this section, we introduce the main features of our graph processing framework.

Overview. Figure 2 shows an overview of our framework running on a cluster of computing nodes connected by a high-speed network. Each node consists of a processing unit with a cached subgraph shard. The processing units are CPUs in our current framework, and can be extended to GPUs or any other graph processing accelerators. Each subgraph shard contains a range of vertices called local vertices, which are a subset of all graph vertices. Boundary vertices with respect to a subgraph shard are vertices from other shards that have edges connecting to the local vertices of the subgraph. Each subgraph shard stores all the associated in/out edges as well as the property of the subgraph. The graph property includes vertex values, and edge weights (if the graph is weighted). Each processing unit computes on its own subgraph shard and updates the graph property iteratively. It is also responsible for sending the values of boundary vertices to other processing units. Structuring the graph processing system this way allows us to decouple computation from communication. We focus on improving the computing efficiency of each processing unit based on its available architecture and resources. Then, we treat all communications as an abstraction of the I/O hierarchy (i.e. memory, disk, and network latency). Note that a subgraph shard does not necessarily need to fit in memory; as a result, the I/O cost may also involve local disk I/O.

3.1 Range-based Graph Partitioning

Graph partitioning is an important step in optimizing the performance of a graph processing system where the input graphs cannot fit in a node's memory. Many system variables such as workload balance, I/O cost etc. are often considered when designing a graph partitioning strategy. There can be different optimal partition strategies depending on the graph structure and application behavior. Moreover, re-partitioning is often required when graphs change, which is costly for large-scale graphs. Our solution is to adopt a lightweight low-overhead partitioning strategy. Our framework deploys a simple range-based partition similar to GraphChi[16], GridGraph [41], Gemini [40] etc. Vertices are assigned to different partitions based on vertex ID, which is re-indexed during graph ingestion. Each partition contains a continuous range of vertices with all associated in/out edges and subgraph properties. To balance the workload, we optimize each partition to contain a similar number of edges. In a p-node system, a given graph $G = (V, E)$ will be partitioned into p continuous subgraphs $G_i = (V_i, E_i)$, where $i = 0, 1, ..., p - 1$. In each G_i, V_i are local vertices and E_i is a set of edges $\{s, t, w\}$, where either source s or destination t belongs to V_i. The rest of the vertices in other partitions are boundary vertices. Assigning all out-going edges of a vertex to the same partition is a way of improving the efficiency of local graph traversals. We also store incoming edges when running graph algorithms such as PageRank.

3.2 Multi-modal Edge-set based Graph Representations

We adopt multi-modal graph representations in our framework to accommodate different access patterns and achieve best data locality for different graph applications. Compressed sparse row (CSR) is a common storage format to store the graph. It provides an efficient way to access the out-going edges of a vertex, but it is inefficient when accessing the incoming edges of a vertex. To address this inefficiency, we choose to store the incoming edges in compressed sparse column (CSC) format, and out-going edges in compressed sparse row (CSR) format.

To improve cache locality, we explore iterative graph computing with an edge-set based graph representation [6, 16, 41]. Similar to the range-based partitioning, each subgraph partition is further converted into a set of edge-sets. Each edge-set contains vertices within a certain range by vertex ID. As shown in Figure 3a, an input graph represented in adjacency matrix format is divided into two partitions, with each partition converted into eight edge-sets. To traverse a graph through out-going edges, we scan the blocked adjacency matrix left to right.

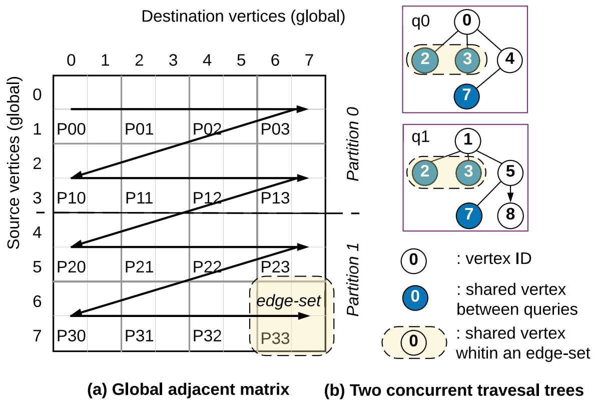


Figure 3: An example of edge-set based graph representation. (a) An input graph is divided into two subgraph partitions, and each partition is converted into 8 edge-sets. To traverse a graph through out-going edges is equivalent to scan the edge-sets in left-right pattern. (b) Graph traversal trees for two concurrent queries. First three levels are presented in the example.

Generating edge-sets is straightforward. We first obtain vertex degrees after partitioning the input graph across machines, and then we divide the vertices of each subgraph into a set of ranges by evenly distributing the degrees. Next, we scan the edge list again and allocate each edge to an edge-set according to the ranges into which source and destination vertices fall. Finally, within each edge-set, we generate the CSR/CSC format using local vertex IDs calculated from global vertex ID and partition offset. The preprocessing reduces the complexity of global sorting, and is conducted in a divide-and-conquer manner.

The granularity of an edge-set is chosen such that the vertex values and associated edges fit into the last level cache (LLC). However, the sparse nature of real large-scale graphs can result in some tiny edge-sets that consist of only a few edges each, if not empty. Loading or persisting many such small edge-sets is inefficient due to the I/O latency. Therefore, it makes sense to consolidate small edge-sets that are likely to be processed together, so that data locality is potentially increased. Consolidation can occur between adjacent edge-sets both horizontally and vertically. The horizontal consolidation improves data locality especially when we visit the out-going edges. Vertical consolidation benefits the information gathering from the vertex's parents.

Concurrent graph traversals can benefit from edge-set representation from two dimensions of locality maintained inside an edge-set: 1) shared neighbor vertices of *frontiers* within an edge-set and 2) shared vertices among queries. A simple example is shown in Figure 3b, where two concurrent queries are presented, each by a graph traversal tree of three levels. Visiting neighbors of vertex 2 and 3 takes just one pass on edge-sets $P_{1i}, i = 0, 1, 2, 3$, and since these two vertices are shared among both queries, query performance can be improved by making only one traversal on these two vertices. The compute engine performs user-defined functions on edges within each edge-set in parallel. Edge-set graph representation also improves cache locality for iterative graph computations like PageRank from two aspects: 1) sequential accesses to edges within a local graph and 2) write locality preserved by storing the edges in CSC format. Updating the vertex value array in ascending order also leads to better cache locality while enumerating the edges in a edge-set.

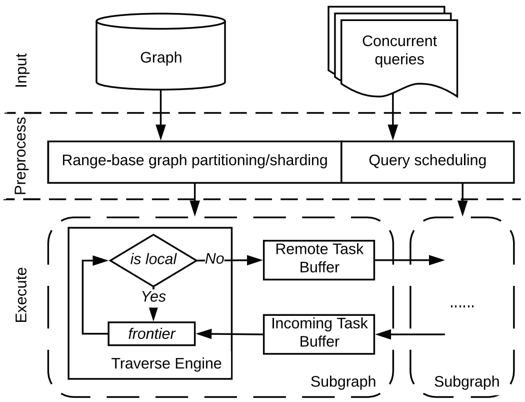


Figure 4: Graph query workflow.

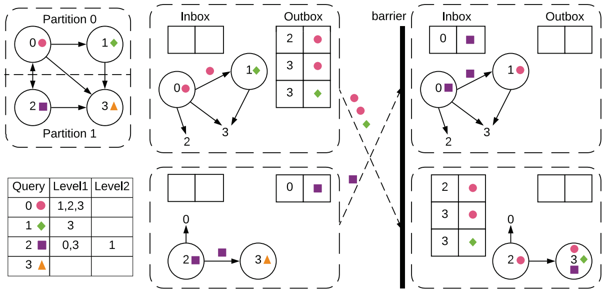


Figure 5: A simple two-partition graph example with four concurrent graph traversals starting from all four vertices. Different queries are distinguished using different symbols. Each partition has an inbox buffer for incoming tasks and an outbox buffer for outgoing tasks. Each task is associated with the destination vertex's unique ID. The visited vertices are synchronized after each iteration and won't be visited.

3.3 Query Processing

Efficient implementation of a distributed graph engine requires balancing computation, communication and storage. Our framework supports both the vertex-centric and partition-centric models. We specifically optimized the partition-centric model to handle graph traversal-based algorithms such as *k-hop*, BFS. The performance of such models depends strongly on the quality of the graph partitions. Figure 4 shows the graph traversal iterations in the partition-centric model (which generally requires fewer supersteps to converge compared to the vertex-centric model). In the partition-based model, vertices can be classified into local vertices and boundary vertices. The values for local vertices are stored in the local partition, while

```
void abstract compute();
void sendTo(V destination, M msg);
void voteTohalt();
bool ifHasVertex(V vid);
bool isLocalVertex(V vid);
bool isBoundaryVertex(V vid);
Collection getLocalVertices();
Collection getBoundaryVertices();
Collection getAllVertices();
void barrier();
```

Listing 1: Partition-centric Model [27]

boundary vertex values are stored in the remote partitions. Local vertices communicate with boundary vertices through messages. A vertex can send a message to any other vertices in the graph using the destination vertex's unique ID.

To illustrate the partition-centric model, we consider two operations: local read and remote write, both of which incur cross-partition communications. Local read is performed when reading the value of a boundary vertex. For example, the PageRank value of a local vertex is calculated from all the neighboring vertices, some of which are boundary vertices. In this case, a locally updated vertex value has to be synchronized across all partitions after each iteration. In other cases, a partition may need to update the value of a boundary vertex of the partition. For example, in subgraph traversals involving traversing depth, when a boundary vertex is visited, its depth needs to be updated remotely. The boundary vertex ID with its value along a traverse operator will be sent to the partition to which it belongs. In that partition, the vertex value will be asynchronously updated and the traversal on that vertex will be performed based on the new depth. In a sense, all vertices are updated locally to achieve the maximum performance through efficient local computation, and all changes of the graph property are exchanged proactively across partitions using high speed network connections. A simple example of subgraph traversal is shown in Figure 5.

Concurrent queries can be executed individually in request order, or processed in batches to enable subgraph sharing among queries. To mitigate the memory pressure in concurrent graph queries, we utilize dynamic resource allocation during graph traversals. We only need to keep vertex values for those in previous and current levels, instead of saving value per vertex during the entire query.

3.4 Programming Abstraction

We next introduce the programming API deployed in our framework. We provide an interface for the partition-centric model, which was first introduced by Giraph++ [27] and has been quickly adopted and further optimized in recent works [25, 34]. Listing 1 shows the interface of the basic methods call in the partition-centric model.

We provide two functions to accommodate different categories of graph applications: a) graph traversal on graph structure and b) iterative computation on graph property. Graph traversal involves data-intensive accesses and limited numeric operations. The irregular data access pattern leads to poor spatial locality and introduces significant pressure on the memory subsystem. Computation on graph property often involves more numeric computation which shows hybrid workload behaviors [20]. The graph traversal pattern is defined in the **Traverse** function, and the iterative computation is defined in **Update** function. An example of *k-hop* implementation is shown in Listing 2.

```
def Traverse(task queue: Q, hops: k) {
  while any s in Q {
    if (s.hops < k) {
        if (isLocalVertex(s)) {
          for (t in s.neighbors and !visited(t)) {
            t.hops = s.hops + 1
            if (isLocalVertex(t)) Q.push(t)
            else sendTo(t, t.hops)
            visited(t) = true
          }
        }
    }
    Q.pop(s)
  }
}
```

Listing 2: k-hop Traversal: For each vertex in a local task queue, visits its neighbors and puts them into two queues based on: local vertices will be inserted into the local task queue, while boundary vertices will be sent to a remote task queue. All neighbors will be marked as visited and shared cross all processing units. The maximum depth of traversal is defined by hops k.

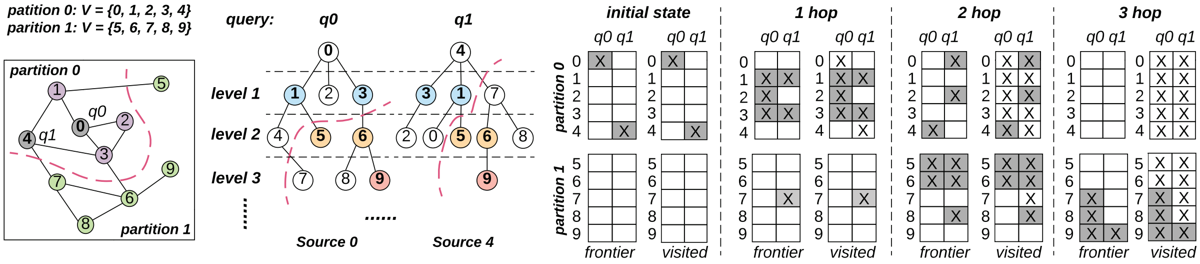
```
def Gather(v, sum) sum += v.val
def Apply(v, sum) v.val = 0.15 + 0.85 * sum
def Scatter(v) v.val / v.outdegree
```

Listing 3: PageRank: The gather phase collects inbound messages. The apply phase consumes the final message sum and updates the vertex data. The scatter phase calculates the message computation for each edge.

The **Update** function is an implementation of the Gather-Apply-Scatter (GAS) model by providing a vertex-programming interface. A PageRank example using the GAS interface is shown in Listing 3. The function looks no different than a normal GAS model graph processing framework. However, our implementation does not generate additional traffic in the gather phase since all edges of a vertex are local.

3.5 Concurrent Queries Optimization

We further optimize the concurrent queries by leveraging several state-of-art techniques. In practice, it is inefficient to use a set or queue data structure to store the *frontier* since the union or set operation is expensive with a large number of concurrent graph traversals. In addition, the dramatic difference in *frontier* size at different traversal levels introduces dynamic memory allocation overhead. It also requires a locking mechanism if the *frontier* is processed by multiple threads. Instead of maintaining a task queue or set, we implement the approach introduced in MS-BFS [26] to track concurrent graph traversal *frontier* and *visited* status, and extend it to distributed environments. For each query, we use 2 bits to indicate if a vertex exists in the current or next *frontier*, and 1 bit to track if it has been visited. A fixed number of concurrent queries are decided based on hardware parameters, for example, the length of the cache line. The *frontier*, *frontierNext* and *visited* are stored in arrays for each vertex to provide constant-time access.



(a) An example graph (b) Two concurrent graph traversal queries (c) *Frontier* and *visited* bit array at each hop.

Figure 6: An example of bit operations for two concurrent queries.

An example graph is shown in Figure 6. A graph with 10 vertices is divided into two machines using range-based partitioning. Partition 0 contains vertices $V : \{0 \sim 4\}$, and partition 1 contains vertices $V : \{5 \sim 9\}$. Each partition maintains a *frontier* and *visited* bit array for each query. Figure 6b shows the traversal tree for each query. In the example, we show two concurrent queries starting from source vertices 0 and 4. Figure 6c shows the bit array representations for *frontier* and *visited* nodes. The *frontier* in the current hop is from *frontierNext* in the previous level. Each row represents a vertex, and each column represents a query. The main idea behind this is queries share same vertices in each iteration, and data locality is preserved if updating concurrent queries at same time.

4 EXPERIMENTAL EVALUATION

To evaluate the efficiency of our system and its optimizations, we measured the system performance using both real-world and semi-synthetic graph datasets. We tested our system with various types of graph algorithms, and reported experimental results on scalability with respect to input graph size, number of machines and number of queries. We compared the performance of our system with open-source graph database Titan [3], and state-of-the-art graph processing engine Gemini [40].

4.1 Experimental Setup

Graph Algorithms. In our experimental evaluation, we used two graph algorithms to show the performance of our system running different types of graph applications.

K-Hop Query is a fundamental algorithm for graph traversals. We use it to evaluate the performance of concurrent queries. Most of our experiments are based on the *3-hop* query, which traverses all vertices in a graph that are reachable within 3 hops from the given source vertex. For each query, we maintain a *frontier* queue and *visited* status for each vertex. Initially all vertices are set as not visited, and *frontier* contains the source vertex. The level of a visited vertex or its parent is recorded as vertex value. The unvisited neighbors of the vertices in the frontier will be added to the *frontier* for the next iteration. The details of the implementation are illustrated in Listing 2. The main factor we used to evaluate the performance of the query system is the response time for each query in a concurrent queries environment. We tested from 10 to 350 concurrent queries, and reported the query time for each query.

Table 1: Datasets Description

Experimental Datasets	Vertices	Edges
Orkut (OR-100M)	3,072,441	117,185,083
Friendster (FR-1B)	65,608,366	1,806,067,135
Friendster-Synthetic (FRS-72B)	131,216,732	72,224,268,540
Friendster-Synthetic (FRS-100B)	984,125,490	106,557,960,965

PageRank is a well-known algorithm that calculates the importance of websites in a websites graph. In PageRank all vertices are active during the computation. The vertex page-rank value is updated after gathering all the neighbors' page-rank values. In our experiments, we ran 10 iterations for performance comparison. An illustration of our implementation using the GAS (Gather-Apply-Scatter) API is shown in Listing 3, with the sum value for each vertex initialized to zero. Although our system is mainly for the concurrent queries, we use PageRank to evaluate the iterative graph computation applications, which have different access patterns compared to graph traversals.

Software and Hardware Configuration. We conducted most of our experiments on a 9 server machines cluster, each has an Intel(R) Xeon(R) CPU E5-2600 v3, having a total of 44 cores at 2.6 GHz and 125 GB main memory. Our system and all algorithms were implemented in C++11, compiled with GCC 5.4.0, and executed on Ubuntu 16.4. We used Socket and MPI (Message Passing Interface) for network communication.

Datasets. In our evaluation, we experimented with both real-world and semi-synthetic datasets. We used two real world graphs: Orkut and Friendster from SNAP [17], and two semi-synthetic graphs: both are generated from Graph 500 generator with Friendster to test the system's ability to process graphs at different scales. Orkut and Friendster are on-line social networks where users form friendships with each other. Orkut has 3 million vertices and 117 million edges with a diameter of 9, while Friendster has 65.6 million and 1.8 billion edges with a diameter of 32. Both graphs form large connected components with all edges. Two semi-synthetic graphs are generated with Graph 500 generator and Friendster graph. Given a multiplying factor *m*, the Graph 500 generator produces a graph

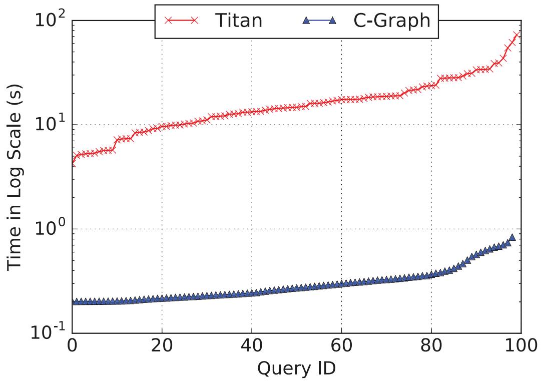
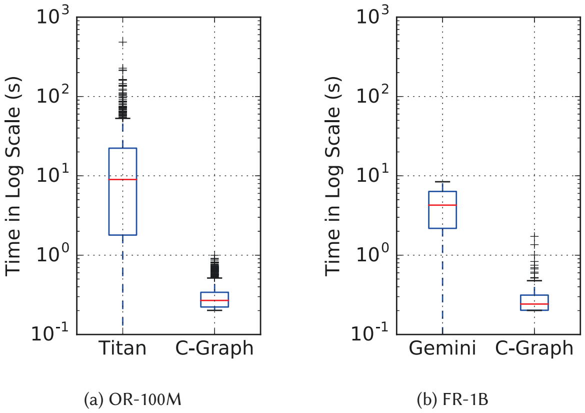


Figure 7: Single machine performance comparison of 100 concurrent *3-hop* queries with Titan running OR-100M graph.



(a) OR-100M

(b) FR-1B

Figure 8: Response time distribution comparison of 100 concurrent *3-hop* queries. (a) Compared with Titan running Orkut (OR-100M) graph on single machine. (b) Compared with Gemini running Friendster (FR-1B) graph on three machines.

having *m* times vertices of Friendster, while keeping the edge/vertex ratio of the Friendster. The smaller semi-synthetic graph has 131.2 million vertices and 72.2 billion edges, and the larger semi-synthetic graph has 985 million vertices and 106.5 billion edges. The details of each graph are shown in Table 1.

4.2 Experimental Results

We used the open-source graph database Titan [3], which supports concurrent graph traversals, as a baseline. Since Titan took hours to load a large graph, we used a small graph Orkut to compare the single machine performance running Orkut on Titan with our system. We used the internal APIs provided by Titan for both graph traversals and PageRank. We also experimented with the well-known open-source graph database Neo4j [22]. However, this system was even slower to load and traverse a large graph. Therefore, we did not include Neo4j in our comparison.

How does Response Time Impact User Experience? Before we dive into the experimental results, we first discuss an important quality metric of an online business like a website or a database: response time. There is a strong correlation between response time and business metrics since wait time heavily impacts user experience. To quantify the performance impact on a query, the following three thresholds have been defined [1, 24]:

- Users view response time as instantaneous (0.1-0.2 second): Users can get query results right away and feel that they directly manipulate data through the user interface.
- Users feel they are interacting with the information (1-5 seconds): They notice the delay, but feel that the system is working on the query. A good threshold is under 2 seconds.
- Users are still focused on the task (5-10 seconds): They keep their attention on the task. This threshold is around 10 seconds. Productivity suffers after a delay above this threshold.

According to the above thresholds, we would reasonably expect a distributed graph processing system to respond to a set of (say, 100–300) concurrent queries within very a few seconds (say 2 seconds).

System Performance. We compared the concurrent *3-hop* query and PageRank performance with the graph database Titan [3] on a single machine. We run 100 concurrent queries for both systems, with each query containing 10 source vertices. The source vertices are randomly chosen, with each system performing 1000 random subgraph traversals to avoid both graph structure and system biases. The average response time for a query is calculated from the 10 subgraph traversals of each query, and average response times for 100 queries are shown in Figure 7, sorted in ascending order.

The results were encouraging, with C-Graph achieving a 21×–74× speedup over Titan. Moreover, our system exhibited a much lower upper bound on query time, with all 100 *3-hop* queries returning within 1 second, while Titan took up to 70 seconds for some queries. In addition, our system showed much lower variation in response time.

We also compared the distribution of all 1000 subgraph traversal times, with the results shown in Figure 8a. The average query response time is 8.6 seconds for Titan, and only 0.25 second for C-Graph. About 10% of the queries in Titan took more than 50 seconds and up to hundreds of seconds. This is likely due to the complexity of the software stack used in Titan, such as the the data storage layers and Java virtual machine. These inefficiencies make the results for PageRank running on Titan even worse. For the Orkut (OR-100M) graph, Titan execution time was hours for a single iteration while C-Graph only took seconds. Overall, our system showed both better and more consistent performance gains compared to Titan.

Most existing graph processing systems lack the ability to handle concurrent queries in large-scale graphs. We use Gemini as an example of how inefficient a design that lacks concurrency can be. Simply using the alternative way instead of re-design the concurrent support, for example making Gemini start with multiple source vertices, will fail. In these systems, concurrently-issued queries are serialized and a query's response time will be determined by any

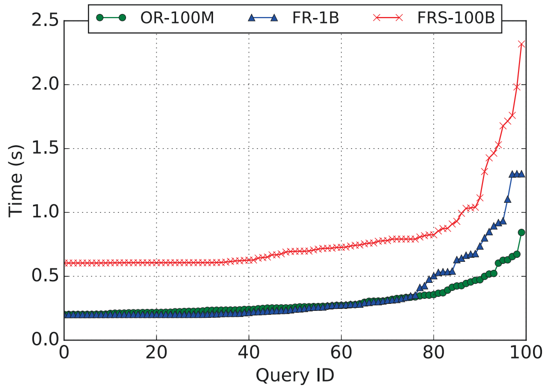


Figure 9: Data size scalability results of response times for 100 concurrent *3-hop* queries.

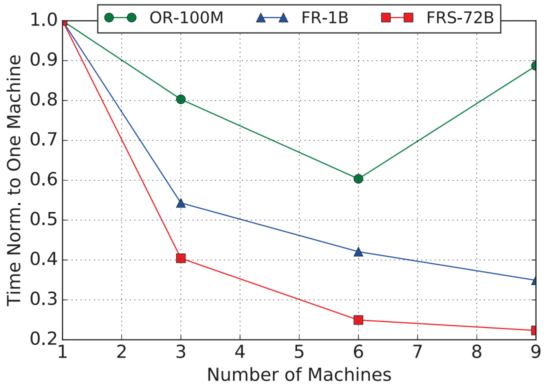


Figure 10: Multi-machine scalability results for PageRank.

backlogged queries in addition to the execution time for current query. We used three machines and repeated the 100 queries with the Friendster (FR-1B) graph on both systems. The response time distribution is shown in Figure 8b. Even though Gemini is very efficient and only takes tens milliseconds for a single *3-hop* query, the average query response time is around 4.25 seconds due to the stacked up wait time. The average response time for C-Graph is only about 0.3 seconds.

Next we focus on the scalability of our system. We ran experiments with different input graph datasets, increasing number of machines and query counts.

Data Size Scalability. For concurrent queries, an important performance indicator is how the upper bound of the response time scales as the input graph size increases. A good query system should guarantee that all queries return within latencies that are acceptable to the users. To understand how our system scales with increased input graph size, we measured its response times for different datasets: Orkut (OR-100M) with 100 million edges, Friendster (FR-1B) with 1 billion edges, and Friendster-Synthetic (FRS-100B) with 100 billion edges.

Figure 9 shows the histogram of response time for 100 concurrent *3-hop* queries running different graphs with 9 machines. We observed that for both graphs, about 85% queries return within 0.4 second for FR-1B, and for FRS-100B the response time slight increases to 0.6 second for the same percentage of queries. The upper bound of query response time is 1.2 seconds for FR-1B, and for FRS-100B it increases slightly to 1.6 seconds. The upper bound of response time for both graphs is within the 2.0 seconds threshold. Note that the response time highly depends on the average degree of root vertices, which is 38, 27, 108 for OR-100M, FR-1B and FRS-100B, respectively.

Multi-machine Scalability. We studied the scalability of our system with an increasing number of machines. We experimented both types of applications: PageRank and concurrent *3-hop* queries.

We examined the inter-machine scalability using 1 to 9 machines to run PageRank on graph datasets OR-100M, FR-1B and FRS-72B. The results are shown in Figure 10. All results are normalized to single machine execution time of corresponding graph. Overall the scalability is very good. For FR-1B graph, it achieves speedup of 1.8x, 2.4x, and 2.9x using 3, 6 and 9 machines, respectively. With more machines the inter-machine synchronization becomes more challenging. In the smallest graph OR-100M, as expected, the scalability becomes poor beyond 6 machines as communication time dominates the execution. We observed better scalability with the largest graph FRS-72B, achieving up to 4.5x speedup with 9 machines.

Figure 11 depicts the response time distribution for 100 concurrent k-hop queries on a single graph using different number of machines. While the machine number increases, most of the queries are able to be completed in a short time, i.e., 80% queries receive a response within 0.2 seconds, and 90% queries finish within one second. For a fixed amount of concurrent traversal queries, as the number of machines used grows up, the number of visited distinct vertices does not vary, while the number of boundary vertices increases significantly. More boundary vertices lead to increased communication overhead for synchronization. In our framework, we employ the partition-centric model combined with the edge-set technique to solve this problem.

Query Count Scalability. The main goal of our framework is to execute concurrent graph queries efficiently. To evaluate this property, we study the scalability of our framework as the query count increases. Figure 12 shows the response time distribution for increasing number of concurrent *3-hop* queries running the FRS-100B graph on 9 machines. Up to 100 concurrent *3-hop* queries, most of the queries can finish in a short time. 80% of the queries are completed within 0.6 seconds, and 90% queries finish within one one second. When the concurrent query count reaches 350, the performance of C-Graph begins to degrade. About 40% queries are able to respond within one second, 60% queries can finish within the 2 seconds threshold. We have to wait 4 to 7 seconds for the remaining queries. The slowdown of the framework is mainly caused by resource limits, especially due to the large memory footprint required for concurrent queries. Since every query returns with found paths, the memory usage increases linearly with the query count.

We further compared the performance and scalability of C-Graph to Gemini in order to maximize the query hops. We experimented

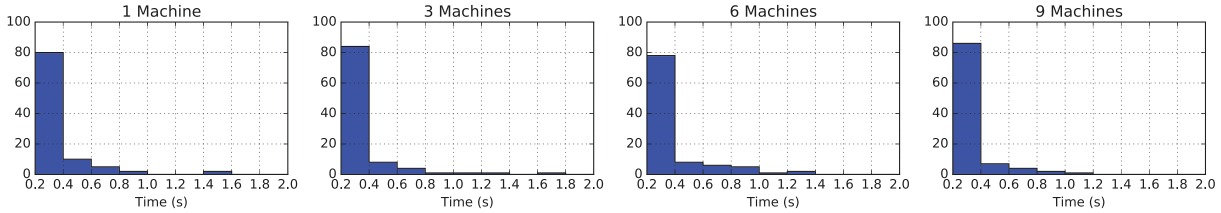


Figure 11: Multi-machine scalability results for 100 queries with FR-1B graph.

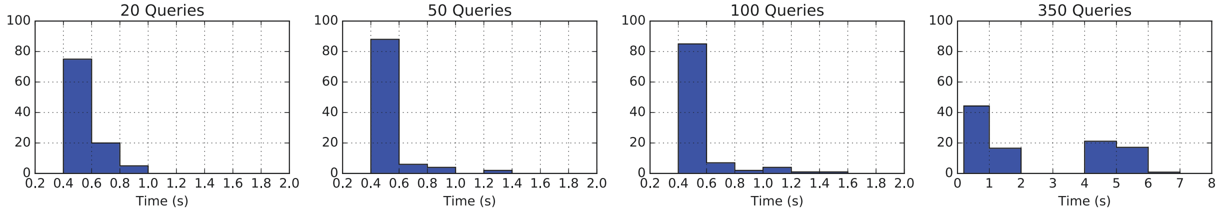


Figure 12: 3-hop query count scalability results for FRS-100B graph.

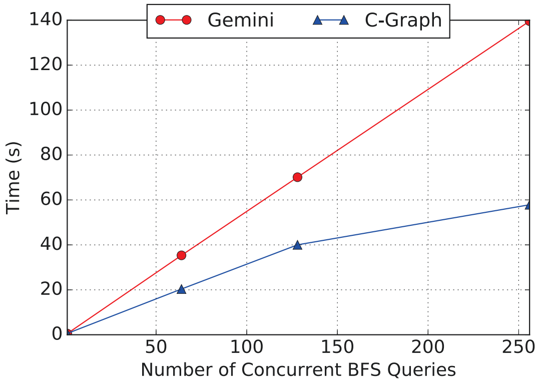


Figure 13: Performance comparison of concurrent BFS queries with Gemini running FR-1B graph on three machines.

with 1, 64, 128 and 256 concurrent BFS queries using the Friendster (FR-1B) graph on 3 machines. Since Gemini doesn't support concurrent queries, we reported total execution time for serialized queries running on Gemini. Also, because our framework reaches the system's memory limit when running higher number of hops with more than 25 concurrent BFS queries, we enabled bit operations in this experiment. As Figure 13 shows, the execution time for Gemini is linear with the number of concurrent BFS queries. C-Graph starts with the same performance for a single BFS which is about 0.5 seconds. However C-Graph execution time increases sublinearly with the number of concurrent BFS queries. As a result C-Graph outperforms Gemini by about 1.7× at 64 and 128 concurrent BFSs, and 2.4× at 256 concurrent BFSs.

5 RELATED WORK

Graph Processing Systems with Query Support. Neo4j [22] and HyperGraphDB [14] focus on supporting online transaction processing (OLTP) on graph data. However, they are not distributed and cannot support web-scale graphs partitioned over multiple machines. Titan [3] supports distributed graph traversals over multiple machines, but its performance is a concern due to the complexity of its software stack. State-of-art graph processing systems often have better performance, however, they generally lack native support for concurrent queries [28, 35, 36, 40]. For example, Gemini [40] is an efficient distributed graph computing system, which outperforms C-Graph in single application performance. However, it cannot handle concurrent queries. Executing the queries serially increases the average response time.

Concurrent Graph Queries. There is an increasing interest in concurrent graph processing including queries for graph processing systems [7, 13, 18, 21, 23, 26, 32, 33]. However, we found that prior works on concurrent queries usually evaluate only small graphs, and don't support distributed environments. MS-BFS [26] introduced level sharing and bitwise operation for efficient multi-source BFS. It works for small-world graphs, and only supports querying in batches and may therefore not be suitable in an interactive multi-user environment. iBFS [18] supports concurrent queries on multi-GPUs. However, iBFS does not partition the graph, it simply copies it on multiple GPUs, and distributes the queries across all of them. The system is limited to graphs that can fit in their entirety in the GPU memory and therefore it only works for small graphs. Wukong [23] is a distributed graph-based RDF store that leverages RDMA-based graph exploration to provide highly concurrent and low-latency queries over large data sets. Congra [21] extended an existing shared-memory graph processing framework and provided a novel scheme for scheduling concurrent graph processing queries on shared memory based systems. It supports more complex graph algorithms involving graph computation so it was only evaluated with small graphs of at most one hundred millions edges.

6 CONCLUSION AND FUTURE WORK

In this paper we presented our work on a concurrent graph processing framework called C-Graph. This system is designed to meet the industrial requirements of efficiently handling a group of simultaneous graph queries on large graphs, rather than accelerating a single graph processing task exclusively on a server/cluster as in prior work. To achieve this goal, the proposed framework maintains global vertex states to facilitate graph traversals, and supports both synchronous and asynchronous communication interfaces. For any graph processing tasks that can be decomposed into a set of local traversals, such as the graph *k-hop* reachability query, our proposed system exhibited excellent performance. In future work, we will look into more architectural optimization of the infrastructure for cloud computing platforms to further improve concurrent graph computing performance, and extend the framework to support more types of graph applications.

REFERENCES

[1] 2011. How Response Times Impact Business? https://calendar.perfplanet.com/2011/how-response-times-impact-business/. (2011).

[2] 2014. Apache Giraph. https://giraph.apache.org/. (2014).

[3] 2014. Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/. (2014).

[4] 2017. JanusGraph Distributed Graph Database. https://github.com/JanusGraph/janusgraph. (2017).

[5] James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. 2012. K-reach: who is in your small world. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1292–1303.

[6] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: an efficient graph processing system on a single machine. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 409–420.

[7] Ayush Dubey, Greg D Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment* 9, 11 (2016), 852–863.

[8] Karthi Duraisamy, Hao Lu, Partha Pratim Pande, and Ananth Kalyanaraman. 2016. High-Performance and Energy-Efficient Network-on-Chip Architectures for Graph Analytics. *ACM Trans. Embed. Comput. Syst.* 15, 4 (2016), 66:1–66:26.

[9] Ioanna Filippidou and Yannis Kotidis. 2015. Online and On-demand Partitioning of Streaming Graphs. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*. 4–13.

[10] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.

[11] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 855–864.

[12] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. (2017). arXiv:arXiv:1706.02216

[13] Matthias Hauck, Marcus Paradies, and Holger Fröning. 2017. Can Modern Graph Processing Engines Run Concurrent Queries Efficiently? (2017).

[14] Borislav Iordanov. 2010. HyperGraphDB: a generalized graph database. *Web-Age information management* (2010), 25–36.

[15] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 1343–1350.

[16] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 8. 31–46.

[17] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1:1–1:20.

[18] Hang Liu, H Howie Huang, and Yang Hu. 2016. iBFS: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 403–416.

[19] Peter Macko, Daniel Margo, and Margo Seltzer. 2013. Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, 18.

[20] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*.

[21] Peitian Pan and Chao Li. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 217–224.

[22] I. Robinson, J. Webber, and E. Eifrem. 2013. *Graph Databases*. O'Reilly Media, Incorporated. http://books.google.com/books?id=RTvAmQEACAAJ

[23] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration.. In *OSDI*. 317–332.

[24] Ben Shneiderman. 1984. Response Time and Display Rate in Human Performance with Computers. *ACM Comput. Surv.* 16, 3 (1984), 265–285.

[25] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A subgraph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*. Springer, 451–462.

[26] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment* 8, 4 (2014), 449–460.

[27] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[28] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[29] Jim Webber. 2012. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 217–218.

[30] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *IEEE Big Data*. 649–658.

[31] Yinglong Xia, Ilie G. Tanase, Lifeng Nai, Wei Tan, Yanbin G. Liu, Jason Crawford, and C-Y. Lin. 2014. Explore Efficient Data Organization for Large Scale Graph Analytics and Storage. In *IEEE Big Data*. 942 – 951.

[32] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. 2017. Processing Concurrent Graph Analytics with Decoupled Computation Model. *IEEE Trans. Comput.* 66, 5 (2017), 876–890.

[33] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 227–238.

[34] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[35] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[36] Da Yan, James Cheng, M Tamer Özsu, Fan Yang, Yi Lu, John Lui, Qizhen Zhang, and Wilfred Ng. 2016. A general-purpose query-centric framework for querying big graphs. *Proceedings of the VLDB Endowment* 9, 7 (2016), 564–575.

[37] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2016. I/O Efficient ECC Graph Decomposition via Graph Reduction. In *PVLDB*. 516 – 527.

[38] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 87–99.

[39] Wen Zhang. 2017. Knowledge Graph Embedding with Diversity of Structures. In *Proceedings of the 26th International Conference on World Wide Web Companion*. 747–753.

[40] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA.*

[41] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.. In *USENIX Annual Technical Conference*. 375–386.