Core Graph: Exploiting Edge Centrality to Speedup the Evaluation of Iterative Graph Queries

Xiaolin Jiang[*]
xjian049@ucr.edu
CSE Department, UC Riverside
USA

Mahbod Afarin[*]
mafar001@ucr.edu
CSE Department, UC Riverside
USA

Zhijia Zhao
zhijia@cs.ucr.edu
CSE Department, UC Riverside
USA

Nael Abu-Ghazaleh
nael@cs.ucr.edu
CSE Department, UC Riverside
USA

Rajiv Gupta
rajivg@ucr.edu
CSE Department, UC Riverside
USA

Abstract

When evaluating an iterative graph query over a large graph, systems incur significant overheads due to repeated graph transfer across the memory hierarchy coupled with repeated (redundant) propagation of values over the edges in the graph. An approach for reducing these overheads combines the use of a small proxy graph and the large original graph in a two phase query evaluation. The first phase evaluates the query on the proxy graph incurring low overheads and producing *mostly* precise results. The second phase uses these mostly precise results to bootstrap query evaluation on the larger original graph producing *fully* precise results. The effectiveness of this approach depends upon the quality of the proxy graph. Prior methods find proxy graphs that are either large or produce highly imprecise results.

We present a new form of proxy graph named the Core Graph (CG) that is not only *small*, it also produces *highly precise* results. A CG is a subgraph of the larger input graph that contains all vertices but on average contains only 10.7% of edges and yet produces precise results for 94.5–99.9% vertices in the graph for different queries. The finding of such an effective CG is based on our key new insight, namely, a small subset of non-zero centrality edges are responsible for determining the converged results of nearly all the vertices across different queries. We develop techniques to identify a CG that produces precise results for most vertices and optimizations to efficiently compute precise results of remaining vertices. Across six kinds of graph queries and four input graphs, CGs improved the performance of GPU-based Subway system by up to 4.48×, of out-of-core disk-based GridGraph system by up to 13.62×, and of Ligra in-memory graph processing system by up to 9.31×.

CCS Concepts: **• Computing methodologies → Parallel computing methodologies; • Information systems → Computing platforms.**

Keywords: iterative graph algorithms, in-memory, out-of-core, GPU, core graph

[*]Both authors contributed equally to this research.

ACM Reference Format:
Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. 2024. Core Graph: Exploiting Edge Centrality to Speedup the Evaluation of Iterative Graph Queries. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3629571

1 Introduction

Graph analytics is employed in many domains (e.g., social networks, web graphs) to uncover insights from connected data. There has been much work resulting in scalable graph analytics systems for GPUs, multicore servers, and clusters [3–6, 9, 11, 13, 15, 16, 20, 23, 25, 27, 28, 30, 34, 36, 38, 43]. Real world graphs are irregular and large. Thus, significant overheads are incurred due to movement of graph across the memory hierarchy and repeated propagation of values over the edges in the graph. These overheads are exacerbated due to the iterative nature of graph analytics. Thus, in spite of the numerous advances, efficient processing of large and irregular graphs remains a challenge.

One approach [18, 42] for dealing with the above challenge employs a *two-phase* (2Phase) query evaluation as shown in Figure 1. Here a small proxy graph corresponding to the large original graph is identified once and then the combination of the proxy graph and original graph is used to evaluate all future queries. *Note that for graphs with a large number*

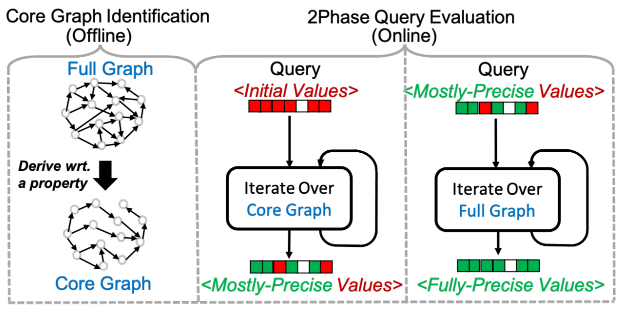


Figure 1. Proxy Graph based 2Phase Evaluation.

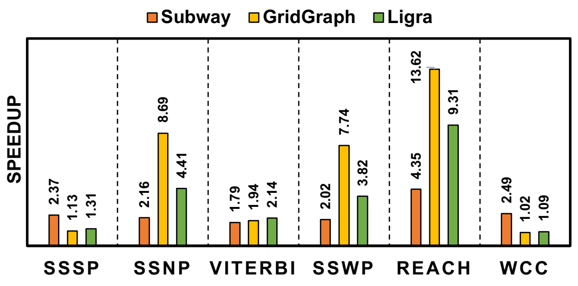


Figure 2. Speedups with CG over without CG for Friendster (FR) [1] input graph with 2.586 billion edges.

of vertices, there is an equally large number of vertex-specific queries (e.g., each vertex can serve as a source of a shortest path query). The first phase evaluates the query on the proxy graph incurring low overheads and producing mostly precise results (mostly green and some red values in Figure 1). Then, the second phase uses these mostly precise results to bootstrap query evaluation on the larger original graph producing fully precise results (all green values in Figure 1). Resuming query evaluation in the second phase from vertices whose property values are impacted in the first phase, guarantees that the 2Phase algorithm will produce correct results for 100% of vertices [18, 42]. Given its generality, this approach is applicable to different kinds of systems – GPU-based Subway, in-memory Ligra, out-of-core GridGraph – as shown in Figure 2. Moreover, the improvements are largely complimentary to other platform specific optimizations incorporated in different graph processing systems. However, for the above approach to be effective, the proxy graph must fulfill two key requirements:

- **RQ1**: The proxy graph should be ***much smaller*** than the original so that the first phase incurs substantially reduced graph transfer overhead and performs little redundant propagation of values over edges; and
- **RQ2**: Convergence over the proxy graph should produce query results that have ***mostly precise***, i.e. most have converged to the same values that are obtained upon convergence over the original graph. Thus, the second phase requires little effort to reach full convergence (i.e., convergence for all vertices).

Prior methods [18, 42], *Reduced Graph* [18] and *Abstraction Graph* [42], are proxy graphs with significant limitations. The first method by Kusum et al. [18] collapses parts of the graph eliminating many vertices that cannot be queried and produces a proxy graph that is too large (roughly 50% of original size [18]). Although the *Abstraction Graph* [42] overcomes the limitations of *Reduced Graph* and produces a small proxy graph, it yields query results that are imprecise – it produces imprecise results for over 53% of the vertices. Similar lack of precision is observed when *graph sampling* is used

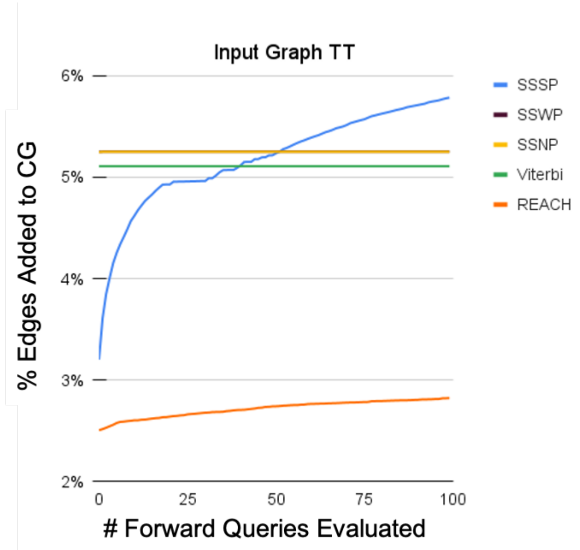


Figure 3. # of Non-Zero Centrality Edges identified with increasing number of queries for Twitter (TT) [7] graph.

Table 1. Average number of queries out of a total of 20 forward queries that select an edge added to CG.

G	SSSP	SSNP	Viterbi	SSWP	REACH
TT	13.01	19.49	20.00	19.99	17.50

to to produce a small proxy graph [21, 29, 41]. *Query-by-sketch* [39] identifies a self-contained subgraph (meta-graph). However, it is limited to *queries that find the shortest path between two vertices.* Our approach handles other kind of queries beside shortest paths and it finds property values from a source vertex to *all destination vertices.*

In this paper we develop a new approach for identifying a proxy graph, called the **Core Graph** (CG), that satisfies both the aforementioned requirements. That is, Core Graph is both small and produces highly precise results. The CG includes all the vertices from the original graph so that any vertex-specific query can be evaluated and it contains only a subset of edges. *Which edges to include in the CG, and how to identify these edges efficiently, is one of our key contribution.*

Consider finding the CG for use by single-source shortest-path (SSSP) queries. We note that the *edge betweenness centrality* (ebc), which is defined as the number of the shortest paths that contain an edge in a graph [24], can be used to identify edges that are important to SSSP queries. Each edge that has non-zero ebc value should be included in the CG as it plays a role in establishing a shortest path between at least a pair of vertices. If all non-zero centrality edges are included in the CG, the graph remains *well connected* via shortest paths, i.e. if there is a path between a pair of vertices in the original graph, then the shortest path is also present in the CG. However, identifying all edges with non-zero ebc values is extremely expensive – it requires computing shortest paths from every vertex to every other vertex.

Our key insight is that most edges with non-zero betweenness centrality can be identified by evaluating a small number of queries corresponding to the highest degree vertices in the graph. We can identify edges that lie along shortest paths found via these queries. That is, each edge $u \rightarrow v$ such that value of vertex v equals value of vertex u plus the weight of edge $u \rightarrow v$, lies along some shortest path. This simple approach is not only inexpensive, since it selects edges along shortest paths, it selects edges with non-zero centrality and edges that provide well connectedness. Figure 3 shows that as we solve increasing number of queries, inclusion of newly discovered edges found to fall on shortest paths causes the number of edges in the CG grow very slowly. That is, vast majority of edges included play an important role in the evaluation of many different queries. Moreover, the data for TT graph in Figure 3 shows that this observation is true across many graph algorithms to a very large degree. Table 1 shows that, when edges are identified using 20 forward queries on TT, most edges are frequently selected for inclusion by majority of the queries – similar behavior was also observed for other input graphs. *Therefore, in rest of the paper we limit the number of vertices queried for identifying CG edges to 20.*

Core graphs are useful for efficiently solving queries over large graphs on different systems as they reduce the following overhead costs: on a GPU where the full graph cannot be held in GPU memory, the overhead of repeated graph transfers between host and GPU memory is substantial; and on a shared-memory out-of-core system where the full graph cannot be held in the memory, the overhead of repeated graph transfers from disk and memory is substantial. Recent research has led to systems with reduced graph transfer overheads for GPU-based [12, 17, 22, 32, 33] and Out-of-Core [20, 31, 37, 42, 43] systems. Nevertheless, Core Graphs can substantially improve performance of Subway [32] for GPUs and GridGraph [43] for out-of-core processing. Even for Ligra [34] where the entire graph is held in memory, Core Graph can significantly reduce graph transfer to on-chip caches. Our experiments show that, in all of the above systems, CG also reduces value propagation over edges yielding reductions in computation performed.

The key contributions of our work are as follows:

- **Core Graph Identification and Exploitation**: We present algorithms for finding a core graph by solving a small set of queries to identify most non-zero centrality edges (§2.1). We exploit CG and present a new optimization that improves the efficiency of the 2phase evaluation while producing 100% precise results (§2.2).
- **Experimental Results** (§3): For the 2.586 billion FR graph, across six kinds of queries, our approach yielded CGs containing 5.42% to 10.45% edges and precise results for 97.1–99.9% vertices. Across six kinds of queries and four large input graphs our approach outperforms Subway [32] by up to 4.48×, GridGraph [43] by up to 13.62×, and Ligra [34] by up to 9.31× for computing precise results for all vertices.

2 Core Graph Identification & Exploitation

In this section we introduce the notion of Core Graph and present an algorithm for its identification. We analyze its effectiveness in terms of its precision and sizes. We also develop algorithms and optimizations to evaluate fully precise results for a given query while also benefiting from Core Graphs to speedup query evaluation.

2.1 Identifying a Core Graph

Consider an input graph, $G(V, E)$, where the edges in E are directed and weighted. An edge from vertex u to vertex v is denoted by $e(u, v)$ and $w(u, v)$ denotes its weight. A directed path from u to v is denoted by $p(u, v)$.

Vertex-Specific Queries. The graph algorithms we consider solve different kinds of *vertex-specific* queries. A vertex query $Q(s)$ originates at the source vertex $s \in V$ and upon its evaluation has computed the *property values* $Q(s).Val(v)$ for all other vertices $v \in V - \{s\}$.

Given a source vertex s and a path $p(s, v)$, the property value of v along the path, denoted as $p(s, v).Val(v)$ is computed from $Val(s)$ and the weights of edges along path $p(s, v)$ using a **propagation** operator $\bigoplus$. For example, given the path $p(s, v) = s \rightarrow u \rightarrow v$, $p(s, v).Val(v)$ is given by $Val(s) \bigoplus w(s, u) \bigoplus w(u, v)$.

For the class of graph queries we consider, given multiple paths $p_1, p_2, \cdots p_n$ from s to v, the property value of v corresponding to query $Q(s)$, denoted as $Q(s).Val(v)$, is computed from $p_i(s, v).Val(v)$ values for all p_i's using a **selection** operator, viz., one of:

$$MIN_i(p_i(s, v).Val(v)) \quad or \quad MAX_i(p_i(s, v).Val(v)).$$

Many graph algorithms fall in this category including the six graph algorithms used in our evaluation.

Edge Centrality and Complete Core Graph. As a consequence of evaluating a query $Q(s)$, it is possible to identify all edges belonging to *solution paths* (e.g., shortest paths). That is, we can identify every edge that belongs to some solution path $p(s, v)$ such that $p(s, v).Val(v) == Q(s).Val(v)$. This is because for any edge $e(a, b)$ belonging to a solution path $p(s, v)$, property values of a and b are related as follows:

$$Q(s).Val(b) \;==\; Q(s).Val(a) \bigoplus w(a, b).$$

All edges that belong to a solution path have non-zero centrality values, i.e. they belong to at least one solution path. For a given graph $G(V, E)$, we introduce the notion of the **Complete Core Graph** that is defined as follows. Given a graph $G(V, E)$, the corresponding **Complete Core**

Graph $cCG(V_c, E_c)$ is a subgraph of G which contains all vertices from G and all *non-zero centrality* edges in G, i.e.,

$$V_c = V; \qquad E_c = \{ \ e(a,b) \ | \ Centrality\ of\ e(a,b) > 0 \ \}$$

Since the above definition states that any edge with *non-zero centrality* is included in *cCG*, it implies that any path $p(x,y)$ for which $p(x,y).Val(y)$ is equal to $Q(x).Val(y)$ in G, is also present in *cCG*. Thus, the evaluation of any query on *cCG* produces results that are identical to those produced by evaluating the query on G.

Note that finding the *cCG* does not require computing the exact centrality value of each edge $e(a,b)$ but rather it simply requires identifying edges with non-zero centrality. If upon solving some query $Q(s)$ we observe that

$$Q(s).Val(a) \ \bigoplus \ w(a,b) = Q(s).Val(b)$$

then edge $e(a,b)$ definitely has non-zero centrality. Nevertheless we observe that it is not practical to identify the Complete Core Graph. By solving a single query $Q(s)$, we can identify only the subset of *non-zero centrality* edges that play a role in computing the solution of query $Q(s)$. However, to identify all non-zero centrality edges, in general all queries must be evaluated. Therefore, next we present a heuristic for finding an *incomplete core graph* that is nevertheless very effective in producing highly precise results.

Our Core Graph Algorithm. Since our goal is to accelerate the solving of queries in the first place, we settle on building an *incomplete* **Core Graph** that is computed by solving a *small set of selected queries* (we found 20 vertices are adequate) and use this graph to speedup the evaluation of all future queries. Henceforth we refer to this incomplete core graph simply as the Core Graph (*CG*) which contains a subset of all non-zero centrality edges. As shown in earlier in Figure 3, when we identify sets of non-zero centrality edges of different queries, there is very large overlap in these sets which implies that most of the non-zero centrality edges being found belong to many soultion paths. We will soon show that our approach for building the *CG* produces exact results for most vertices, further confirming the inclusion of most non-zero centrality edges in *CG*, while the exact results for remainder of vertices require further computation using the original full graph to account for the non-zero centrality edges that are missing from *CG* and belong to some solution path for the query being evaluated.

– *Forward and Backward Queries.* Our work is aimed at large graphs with *power law degree distribution.* For such graphs, it is known that high degree vertices are good proxies for high centrality vertices [10]. Thus, a small number of highest degree vertices are used to identify edges with non-zero centrality. Given a chosen high-degree vertex h, we can find high centrality edges by solving query $Q(h)$ and then testing each edge for non-zero centrality. In a directed graph, we actually solve two queries corresponding to each chosen high-degree vertex h: $Q(h, forward)$ and $Q(h, backward)$.

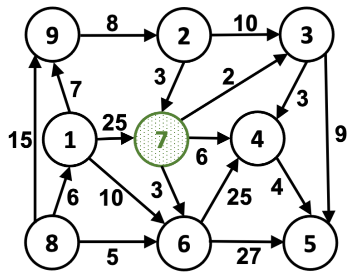
Algorithm 1 Finding *CG* wrt high degree vertices in H.

```
1: Input: Graph $G(V,E)$ and High-Degree Vertex Set $H$
2: Output: $CG(V,E_c)$, $E_c$ contains edges chosen from $E$
3: for each $h \in H$ do
4:     $E_c^f(h)$ = IDENTIFY ( $G(V,E)$, DIRECTION $f$ )
5:     $E_c^b(h)$ = IDENTIFY ( $G^R(V,E^R)$, DIRECTION $b$ )
6:     $E_c = E_c \cup E_c^f(h) \cup E_c^b(h)$
7: end for
8: for all $v \in V$ do
9:     if OutDegree(v)$\neq 0 \wedge$ OutEdges(v) $\cap E_c(h) = \phi$ then
10:        Add an out edge of $v$ to $E_c(h)$
11:    end if
12: end for
13: function IDENTIFY ( $G(V,E)$, DIRECTION $d$ )
14:     Evaluate Query $Q(s)$ on $G(V,E)$
15:     for all $e(u,v) \in E$ do
16:         if $Q(s)$ updates $Q(s).Val(u)$ then
17:             if ($Q(s).Val(u) \bigoplus w(u,v) = Q(s).Val(v)$) then
18:                 if ($d == f$) then
19:                     $E_c(h) = E_c(h) \cup \{ \ e(u,v) \ \}$
20:                 else ▷ ( $d == b$ )
21:                     $E_c(h) = E_c(h) \cup \{ \ e(v,u) \ \}$
22:                 end if
23:             end if
24:         end if
25:     end for
26: end function
```

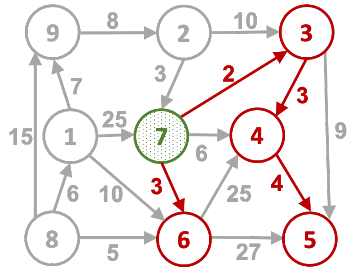
The forward query identifies non-zero centrality edges that lie along paths originating at h and leading to some other vertex. The backward query identifies non-zero centrality edges that lie along paths originating at some other vertex and leading to h. **By computing both forward and backward queries we are able to preserve pairwise reachability among vertices, via h, to a very large extent and thus producing a well connected *CG*.**

– *Additional Connectivity Edges.* Once non-zero centrality edges corresponding to a small set of high degree vertices have been found, we ensure that every vertex with non-zero out-degree has at least one edge included in the core graph to make the graph well connected. **If no outgoing edge is included, we add one**. For SSSP (SSWP) lowest (highest) weight edges are chosen as they are more likely to belong to shortest (widest) paths. This approach ensures that each vertex is connected to the *CG*.

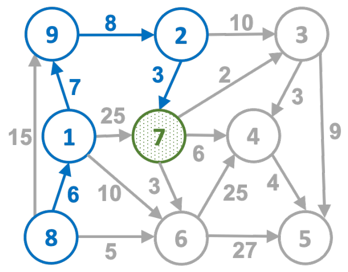
Algorithm 1 shows the above computation. The algorithm repeatedly uses different high degree vertices in H to find additional non-zero centrality edges. Finally, it ensures that at least one out edge of each vertex with non-zero out degree is added to E_c. Note that, when the incomplete Core Graph as constructed above is used to solve a new query, it will produce exact results for some vertices but not for all vertices. Next we illustrate the above algorithm and observations.


(a) **Full Graph** (G).

$7 \longrightarrow *$	1	2	3	4	5	6	7	8	9
7	∞	∞	2	5	9	3	0	∞	∞


(b) **Non-Zero Centrality Edges for** $SSSP(7, forward)$.

$* \longrightarrow 7$	7
1	18
2	3
3	∞
4	∞
5	∞
6	∞
7	0
8	24
9	11


(c) **Non-Zero Centrality Edges for** $SSSP(7, backward)$.

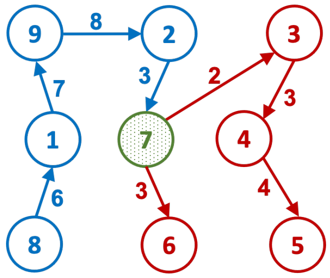

(d) **Core Graph** derived from (b) and (c).

Figure 4. Illustration of Alg. 1 Starting from Full Graph.

Example of Finding Core Graph. We build a core graph for the shortest path problem for the example graph in Figure 4(a). The red and blue edges in Figures 4(b) and (c) are non-zero centrality edges found by solving queries SSSP(7,forward) and SSSP(7,backward). The core graph obtained by combining the two is shown in Figure 4(d). The

Table 2. All Shortest Paths Found: Using the G with 17 Edges (Top) vs. Core Graph CG with 8 Edges (Bottom).

G	1	2	3	4	5	6	7	8	9
1	0	**15**	**20**	**23**	**27**	**21**	**18**	∞	**7**
2	∞	0	**5**	**8**	**12**	**6**	**3**	∞	∞
3	∞	∞	0	**3**	**7**	∞	∞	∞	∞
4	∞	∞	∞	0	**4**	∞	∞	∞	∞
5	∞	∞	∞	∞	0	∞	∞	∞	∞
6	∞	∞	∞	**25**	**27**	0	∞	∞	∞
7	∞	∞	**2**	**5**	**9**	**3**	0	∞	∞
8	**6**	**21**	**26**	**29**	**32**	**5**	**24**	0	**13**
9	∞	**8**	**13**	**16**	**20**	**14**	**11**	∞	0

CG	1	2	3	4	5	6	7	8	9
1	0	**15**	**20**	**23**	**27**	**21**	**18**	∞	**7**
2	∞	0	**5**	**8**	**12**	**6**	**3**	∞	∞
3	∞	∞	0	**3**	**7**	∞	∞	∞	∞
4	∞	∞	∞	0	**4**	∞	∞	∞	∞
5	∞	∞	∞	∞	0	∞	∞	∞	∞
6	∞	∞	∞	∞	∞	0	∞	∞	∞
7	∞	∞	**2**	**5**	**9**	**3**	0	∞	∞
8	**6**	**21**	**26**	**29**	**33**	**27**	**24**	0	**13**
9	∞	**8**	**13**	**16**	**20**	**14**	**11**	∞	0

Algorithm 2 Finding CG for an Unweighted Graph.

```
Input: Original Graph $G(V, E)$ and Query Set $S$
Output: High Centrality Edges $E_c$

QID[*] ← 0 for each vertex
$E_c^f \leftarrow E_c^b \leftarrow \phi$
for $s \in S$ do
    $E_c^f = E_c^f \cup$ TRAVERSE ( $s, G(V, E)$ )
    $E_c^b = E_c^b \cup$ TRAVERSE ( $s, G^R(V, E^R)$ )
end for
$E_c = E_c^f \cup$ REVERSE ( $E_c^b$ )

function TRAVERSE ( $s, G(V, E)$ )
    $CGE = \phi$
    FIFO.PUSH($s$);
    while ! FIFO.EMPTY() do
        $u \leftarrow$ FIFO.POP()
        for all $e(u, v) \in Graph.Outedges(u)$ do
            if QID(v) $\neq s$ then
                ▷ add $e(u, v)$ to $CGE$
                $CGE = CGE \cup \{ e(u, v) \}$
                if QID(v)=0 then
                    FIFO.PUSH(v)
                    QID(v) ← $s$
                end if
            end if
        end for
    end while
    return ( $CGE$ )
end function
```

Table 3. #Edges, #Vertices, and In-Memory Size of Graphs from SNAP [2]: Friendster – FR; Twitter – TT; Twitter – TTW; PokeC – PK. Algorithms: SSSP-single source shortest path; SSNP-single source narrowest path; Viterbi; SSWP-single source widest path; REACH-reachability from a single source; and WCC-weakly connected component which uses *CG* as REACH.

G	| E |	| V |	G Size (MB)	CG Size (MB)				
				SSSP	SSNP	Viterbi	SSWP	REACH
FR [1]	2,586,147,869	68,349,467	20,963	2,436	1,777	1,789	1,777	834
TT [7]	1,963,263,821	52,579,683	15,916	1,680	1,421	1,425	1,425	762
TTW [19]	1,468,365,182	41,652,231	11,914	1,353	1,784	1,146	1,762	656
PK [35]	30,622,564	1,632,804	252	60	51	36	51	21

Table 4. % of Total Edges in the Specialized and General Core Graphs Computed from 20 High-degree Vertices. Overall average is 10.7%.

CG	SSSP	SSNP	Viterbi	SSWP	REACH
FR	10.45%	7.27%	7.33%	7.27%	5.42%
TT	9.36%	7.71%	7.73%	7.71%	7.02%
TTW	10.10%	13.77%	8.34%	13.58%	8.34%
PK	21.85%	18.05%	12.14%	18.18%	12.13%

Table 5. Average % of Vertices for which CG Produces Precise Results for 10 Queries.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	97.1%	99.9%	99.9%	99.9%	99.9%	99.4%
TT	99.6%	99.9%	99.9%	99.9%	99.9%	99.9%
TTW	99.4%	99.9%	99.9%	99.9%	99.9%	98.7%
PK	94.5%	99.9%	99.9%	99.9%	99.9%	99.3%

two tables in Table 2 show results of all shortest path queries computed using the original graph (G) and the core graph (CG). Note that most of the results in the two tables are identical. Only four results shown in red do not match.

First, the reachability for vertex pairs $(6, 4)$ and $(6, 5)$ is present in G but not in CG causing the query SSSP(6), when computed using CG, to result in values of vertices 4 and 5 to be ∞. *However, since no outgoing edge for vertex* 6 *is present, we will add the lowest weight outgoing edge from* 6 *to* 4 *to the core graph.* This will cause the values for vertices 4 and 5 to change to 25 (precise) and 29 (imprecise).

Second, though reachability for pairs $(8, 5)$ and $(8, 6)$ is satisfied by both graphs, in CG the lengths of the paths is longer than the shortest paths in G. Since CG contains a subset of G's edges, shortest path length for a vertex pair computed using G can only be shorter than for CG.

CG for Unweighted Graphs Next we present a heuristic for building a core graph for evaluating queries on *unweighted graphs*. Examples of queries that fall in this category include Reachability–REACH and Weakly Connected Components–WCC. Since such queries rely on reachability, we can identify non-zero centrality edges via forward and backward breadth-first-trees corresponding to a set of high-degree vertices. Next, we describe an algorithm for finding a small core graph that captures reachability characteristics of an unweighted graph.

When constructing a core graph that captures reachability via forward and backward BFS-traversals, same edges can be chosen by traversals originating at different high-degree vertices to the extent possible in order to produce smaller core graphs. Algorithm 2 takes advantage of this sharing in identifying core graph edges. It maintains QID(v) containing the id of the high-degree query vertex that is the first to add an incoming CG edge of v in the set of core graph edges CGE. When an edge $e(u, v)$ is encountered whose source and destination vertex QID's are different, the edge is added to the graph but all core graph edges emanating from v onward are reused by queries labeling vertices u and v.

Studying the Precision and Sizes of Core Graphs. We carried out a study based upon five different kinds of queries and four graphs to evaluate the effectiveness of core graphs. The kinds of queries include SSSP–shortest path, SSNP–narrowest path, Viterbi, SSWP–widest path, and REACH–reachability. The graphs are described in Table 3.

– *Core Graph Sizes.* In Table 4 we present the percentage of all edges that are present in specialized core graphs for SSSP, SSNP, Viterbi (where lower weight edges are more important), and SSWP where higher weight edges are more important. Finally in REACH, weights play no role. For the three large input graphs – FR, TT, TTW – the core graphs contain 7.27% to 13.77% of total edges. For the smaller PK graph this number is higher. **The average over all core graphs found is 10.7%.**

– *Precision of Results.* Though the core graph contains a very small fraction of edges from the full graph, it truly captures its essence. When we evaluated ten random queries for every combination of graph algorithm and input graph, **we found that on average for 94.5–99.9% of the vertices the core graph produces precise results** (i.e., same result as the one produced by the full graph). This data is given in Table 5. Across four kinds of queries (SSNP, Viterbi, SSWP, REACH) CG generated imprecise results for only a tiny number of vertices – a maximum of 310, 40, 36, and 79 vertices for FR, TT, TTW, and PK respectively. For SSSP the fraction of vertices with imprecise results is the highest with average percentage errors in the values for these vertex values being 2.27%, 6.35%, 5.71%, and 3.79% for FR, TT, TTW, and PK. Finally, note that we also provide precision of WCC which is computed using the REACH's *CG*.

Limitations. The above observations hold for irregular graphs with power-law distribution. For other kinds of graphs, core graphs may have different forms and different degree of precision. Also, we have examined six graph properties, there may be other properties for which high precision may be difficult to attain. Finally, as mentioned earlier, our work considers monotonic algorithms for vertex-specific queries. In the remainder of the paper we present an algorithm that exploits core graphs to speedup performance of multiple graph processing systems. However, this algorithm was applied to monotonic graph algorithms. Successful use of core graphs in context of non-monotonic algorithms such as PageRank remains an open problem.

2.2 Exploiting Core Graphs in Query Evaluation

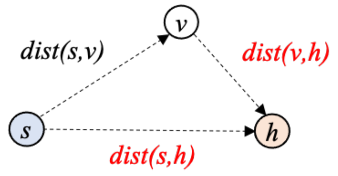
The exact evaluation of a query results requires a two phase approach where the first phase evaluates the query on the small in-memory core graph (CG) and then uses the results obtained to bootstrap the evaluation of the query on the full graph (G). Starting from all vertices whose values are impacted in the first phase, the second phase resumes propagation of values over the full graph to obtain precise results for all vertices. Since most results are computed precisely in the first phase efficiently using the small CG, the work performed during the second phase is greatly reduced.

Next we present a new optimization over the above approach to improve the efficiency of the second phase. The key idea behind this optimization is as follows. After the first phase has completed, we introduce a step that is able to identify some (but not all) of the vertices whose values are already precise and hence will not change in the second phase. For each such vertex v, the incoming edges of v are removed from the full graph G because any propagation via these edges will not change the value of vertex v and hence would be wasteful. Next we present a theorem that allows us to identify some (but not all) vertices with stable values following first phase. We first present the above results in context of the shortest path problem and later show that these results apply to many other graph algorithms.

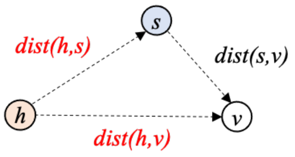
Given a full graph $G(V, E)$ and a high degree vertex h in V such that forward and backward *shortest path* queries $SSSP_f(h, G)$ and $SSSP_b(h, G)$ are evaluated to identify the corresponding core graph $CG(V, E_c)$. Now consider the first phase evaluation of user query $SSSP_f(s, CG)$ on $CG(V, E_c)$ that computes, for each vertex v reachable from s, the shortest path length $dist(s, v).CG$. The following theorem provides the condition under which $dist(s, v).CG$ is precise.

Theorem 1: The computed value $dist(s, v).CG$ is *precise* if one of the following conditions is true:
(a) $dist(s, v).CG == dist(s, h).G - dist(v, h).G$
(b) $dist(s, v).CG == dist(h, v).G - dist(h, s).G$

Proof: To prove the above, we rely on the triangle inequality over the shortest path property as given in [14].



According to the triangle quality for shortest path property:
$dist(s, v).G + dist(v, h).G \geq dist(s, h).G$
$or\ dist(s, v).G \geq dist(s, h).G - dist(v, h).G$ (1)
Since CG is a subgraph of G:
$dist(s, v).CG \geq dist(s, v).G$ (2)
Therefore from (1) and (2) we conclude that:
$dist(s, v).CG \geq dist(s, h).G - dist(v, h).G$
Thus, **if** we observe that
$dist(s, v).CG == dist(s, h).G - dist(v, h).G$ (a)
then $dist(s, v).CG$ must be precise.



Similarly, we use another triangle inequality as follows:
$dist(h, s).G + dist(s, v).G \geq dist(h, v).G$
$or\ dist(s, v).G \geq dist(h, v).G - dist(h, s).G$ (1)
Since CG is a subgraph of G:
$dist(s, v).CG \geq dist(s, v).G$ (2)
From (1) and (2) we conclude that if:
$dist(s, v).CG \geq dist(h, v).G - dist(h, s).G$
Therefore, **if** we observe that
$dist(s, v).CG == dist(h, v).G - dist(h, s).G$ (b)
then $dist(s, v).CG$ must be precise. □

As shown in [14], the graph triangle inequality abstraction given below applies to many different graph properties.
$$property(v_1, v_2) \oplus property(v_2, v_3) \geq property(v_1, v_3)$$
Here $\oplus$ depicts an abstract addition and $\geq$ represents an abstract greater than or equal operator. While operators vary across the different algorithms, the proposed optimization applies many graph algorithms such as widest path, narrowest path, breadth-first search, and others.

Algorithm 3 shows the 2Phase evaluation of a query on a GPU. In the *initialization* step (lines 3 to 8) of this algorithm, the host receives the query vertex and initializes the vertex values such that the values of the outgoing neighbors are computed using the source vertex value and these outgoing neighbors form the initial frontier. Next the host transfers

Algorithm 3 2Phase Algorithm for evaluating query for source vertex s in input graph $G(V, E)$.

```
1:  Input: $s$, $G(V, E)$, and $CG(V_c = V, E_c)$
2:  Output: Query Result – $Val_s(*)$
3:  ▷ Initialization: Initialize $Val$ Array on Host
4:  $Val_s(s) \leftarrow SourceInitVal$
5:  $\forall v \in OutNeighbors(s), Val_s(v) \leftarrow Val_s(s) \bigoplus w(s, v)$
6:  $\forall v \in V - OutNeighbors(s), Val_s(v) \leftarrow InitVal$
7:  Transfer from Host to GPU:
8:      $Val_s(*)$, $OutNeighbors(s)$, & $CG(V_c, E_c)$
9:  ▷ Core Phase: Process Core Graph on GPU
10: ACTIVE ← $OutNeighbors(s)$
11: while ACTIVE $\neq \phi$ do
12:     ACTIVE ← PROCESS ( ACTIVE, $CG(V_c, E_c)$ )
13: end while
14: ▷ Completion Phase: Process Full Graph on GPU+Host
15: ACTIVE ← Impacted Vertices in $V$
16: while ACTIVE $\neq \phi$ do
17:     ACTIVE ← PROCESS ( ACTIVE, $G(V, Reduced(E))$ )
18: end while
19:
20: ▷ Push $Val_s$'s of Vertices in ACTIVE Over outEdges
21: function PROCESS ( ACTIVE , Graph )
22:     NEWACT ← $\phi$
23:     for all $u \in$ ACTIVE do
24:         for all $e(u, v) \in Graph.outEdges(u)$ do
25:             if $Needed(u, v)$ then
26:                 $change$ ← EDGEFUNCTION ( $e(u, v)$ )
27:                 if $change \vee$ FIRSTPHASE2VISIT($v$) then
28:                     NEWACT ← NEWACT ∪ { $v$ }
29:                 end if
30:             end if
31:         end for
32:     end for
33:     return NEWACT
34: end function
```

the initial value array, the frontier, and the core graph to the GPU to begin query evaluation.

On the GPU, the two phases of query evaluation are: *Core Phase* (lines 9 to 13) and *Completion Phase* (lines 14 to 18). During the Core phase, the query evaluation is carried on the much smaller CG and when this phase stabilizes, the second phase begins. In the Completion phase, starting from the vertices that are impacted in first phase, and this time using G, with incoming edges of all the precise vertices removed by Reduced(E), once query is evaluated till the values stabilize and the final result of the query becomes available.

In the completion phase, all reachable vertices must be visited at least once to ensure that their values are push along outgoing edges in FG that were excluded from CG. This is achieved by ensuring that upon first visit to a vertex in this phase, it is always added to the frontier even if its property value has not changed (call to FIRSTPHASE2VISIT(), line 27).

The first phase is fast and effective as it is an in-memory phase which produces precise results for over 94% of the vertices. The second phase is efficient because most needed computations have already been completed and the edge function with atomic operation to propagate values is applied for incoming edges of a small number of vertices.

Note that the 2Phase algorithm is general and can be used to enhance the performance of range of existing systems including out-of-core systems like GridGraph [43] and even in-memory systems like Ligra [34]. In context of in-memory system like Ligra, the Core Phase reduces overall computation performed while in the context of an out-of-core system like GridGraph an in-memory Core Phase reduces the cost of both computation and I/O performed during the second out-of-core phase. For simplicity, we maintain separate core graph and full graph representations and simply switch from core graph to full graph when we transition to second phase.

Table 6. Push Operations for Four Algorithms. Here, CASMIN(a; b) sets a = b if b < a atomically using compare-and-swap; CASMAX is similarly defined.

Alg	NEEDED $(e(u,v))$ / EDGEFUNCTION $(e(u,v))$
SSWP	$Val(v) < min(Val(u), wt(u,v))$ $CASMAX(Val(v), min(Val(u), wt(u,v)))$
SSNP	$Val(v) > max(Val(u), wt(u,v))$ $CASMIN(Val(v), max(Val(u), wt(u,v)))$
SSSP	$Val(v) > Val(u) + wt(u,v)$ $CASMIN(Val(v), Val(u) + wt(u,v))$
Viterbi	$Val(v) < Val(u)/wt(u,v)$ $CASMAX(Val(v), Val(u)/wt(u,v))$
REACH	$Val(v) < Val(u)$ $CASMAX(Val(v), Val(u))$
WCC	$Val(v) > Val(u)$ $CASMIN(Val(v), Val(u))$

3 Experimental Evaluation

We evaluate the benefits of core graphs in improving the performance of query evaluation for the following systems:

- **Subway** [32] system's synchronous algorithm for evaluating queries with reduced data transfer on a GPU;
- **GridGraph** [43] disk-based out-of-core graph processing system with 8GB available memory exceeded by all graph sizes and 4×4 grid partitioning; and
- **Ligra** [34] in-memory graph processing system with push-based algorithms.

The evaluation employs Specialized CGs for SSSP, SSNP, Viterbi, and SSWP while General CG is used to evaluate reachability (REACH) and Weakly Connected Components (WCC). The CGs were derived from evaluation of queries for 20 highest degree vertices in each graph. The choice of 20 was made after it was observed that evaluation of additional queries contributed very few new edges to the CG. This behavior is shown in Figure 3. Four input graphs

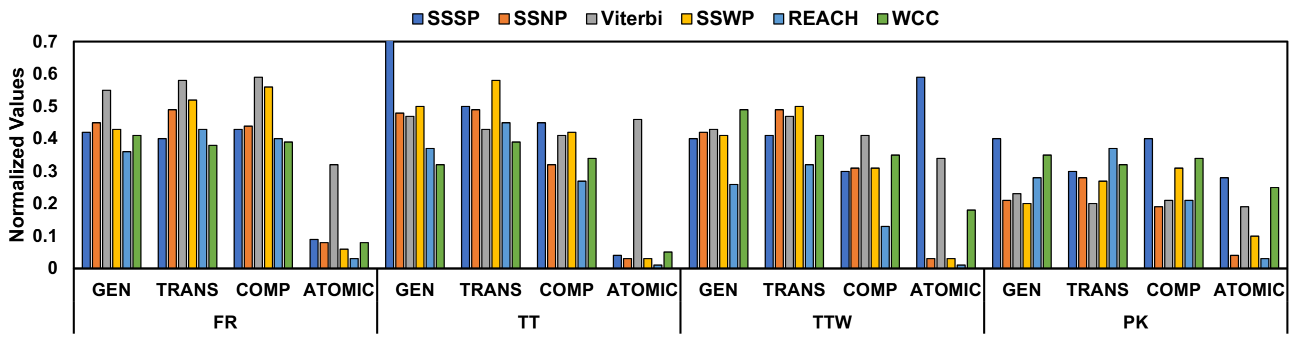
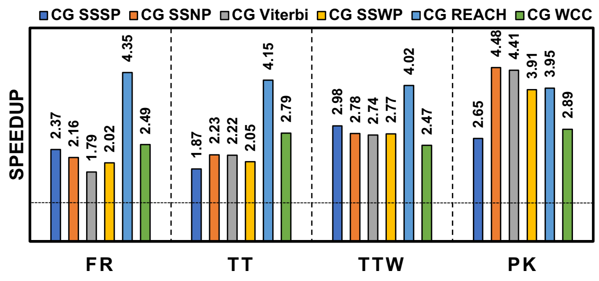


Figure 5. Benefit of CG to **Subway** [32] – 2Phase Values Normalized to Subway: Graph Generation Time - GEN; Data Transfer Time - TRANS; Computation Time - COMP; and # of Push Atomic Updates - ATOMIC.

Table 7. Average Execution Times in Seconds for Core Graph based 2Phase **Subway** [32] across 10 Queries.

G	Specialized CGs				General CG	
	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	9.77s	6.89s	10.54s	8.68s	2.12s	5.79s
TT	9.55s	5.54s	6.85s	6.97s	2.22s	4.01s
TTW	9.91s	8.03s	16.3s	8.25s	1.40s	5.61s
PK	0.19s	0.14s	0.24s	0.16s	0.03s	0.09s



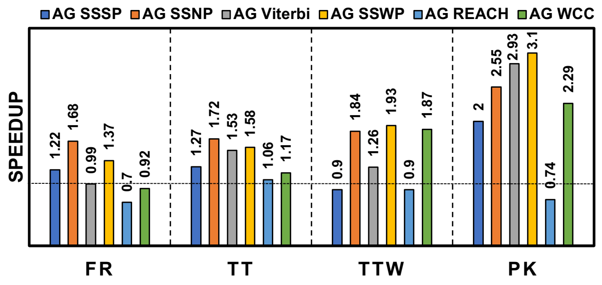


Figure 6. Speedups Over **Subway** Due To Bootstrapping Initial Result from CG and AG.

from Table 3 are used. The default weight generation tool from Ligra is used to generate weights ranging from 1 to the $log(n) + 1$ (where, $n = |V|$).

Experiments were run on NVIDIA Tesla K80 GPU and a 16-core server with AMD Opteron(tm) Processor 6376 and 256GB memory, running on CentOS 7.9. The baselines are the original Subway, GridGraph, and Ligra systems and same settings are used for CG-based 2Phase runs. We perform in-memory evaluation of the query on the core graph in the first phase. The in-memory evaluation on a GPU can be carried out using any of the existing algorithms [6, 13, 15, 16, 28, 38], in our experiments we use [28]. For GridGraph first phase is performed over unpartitioned graph. Data presented represents averages based upon execution of ten random queries for each graph and algorithm combination.

For the largest FR graph the one time cost of identifying the core graphs using Subway [32] for the most (least) expensive Viterbi (REACH) queries is around 14 (7) minutes.

3.1 Speedups over Subway on a GPU

Table 7 and Figure 6 present the executions times of the 2Phase and corresponding speedups over Subway respectively. The time for first phase is no more that 8% of the total query evaluation time. Our 2Phase approach enabled by use of core graphs delivers consistent speedups across all input graphs and all algorithms. For first four benchmarks that depend upon edge weights speedups range from 4.48× to 1.79× and for the last two that do not use edge weights the speedups range from 4.35× to 2.47×. We would also like to point out that when we ran a 2Phase Subway using the Abstraction Graphs [42], as Figure 6 shows, significantly smaller small speedups or even small slowdowns are observed due to AGs low precision.

These performance benefits are due to reductions in graph generation (GEN), data transfer (TRANS), and computation (COMP) – percentage reductions are given in Figure 5. Since the first phase involves in-memory processing, it does not incur graph generation cost and only one time cost of loading the small CG. Because the first phase produces precise results for most of vertices (recall Table 5), the processing of edges in the second phase is reduced. For the first four benchmarks we observe substantial reductions (over 50% in most cases) in all three categories and hence consistent speedups, 4.48–1.79× (2.51× mean), are observed across all benchmarks and graphs. For the last two benchmarks, since most edges are processed once, the reductions in the costs of graph generation and data transfer is typically smaller in comparison to reduction in computation cost. Thus, lower speedups of 2.89–1.31× (2.02× mean) are observed. Although our primary objective was to demonstrate the effectiveness of core graphs in reducing GEN and TRANS overheads, we observe that atomic updates are also reduced significantly (ATOMIC in Figure 5). The reason for these reductions is as follows. In the first phase fewer atomic updates are needed because CG has fewer edges. In the second phase fewer edges function updates are performed because values at nearly all vertices are already precise (recall data in Table 5).

3.2 Speedups over GridGraph and Ligra

We also evaluated the benefits of core graph based 2Phase approach on a non-GPU shared-memory platform. In particular, we considered the GridGraph [43] out-of-core system where a partitioned graph is held on disk and Ligra [34] in-memory system where the entire graph is held in memory.

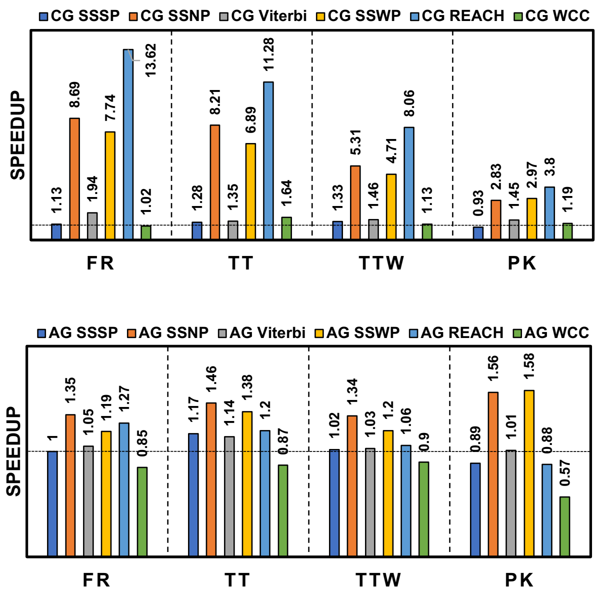


Figure 7. Speedups Over **GridGraph** Due To Bootstrapping Initial Result from CG and AG.

For GridGraph, the first phase of computation is performed in-memory after loading the CG from disk and then the second phase performs partition-based processing. The active frontier for second phase is set to all the vertices whose values have been changed by the first phase. This ensures that maximal amount of updates are performed in the first iteration. This policy generally leads to fewer iterations. During the second phase disk IO savings come due to two reasons: fewer iterations may be performed compared to baseline GridGraph; and during an iteration blocks with no active edges may arise more frequently and hence their fetch from disk will be skipped due to the *selective scheduling* optimization in GridGraph.

Table 8. Average Execution Times in Seconds for Core Graph based 2Phase **GridGraph** [43] across 10 Queries.

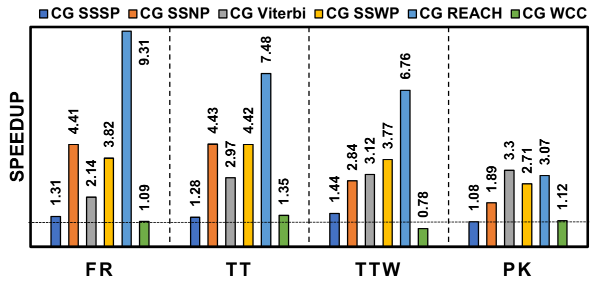
G	Specialized CGs				General CG	
	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	274.4s	39.0s	200.5s	39.1s	11.0s	98.8s
TT	116.3s	24.8s	163.5s	23.2s	7.1s	24.2s
TTW	78.5s	27.4s	108.2s	29.8s	6.5s	34.4s
PK	3.2s	1.5s	3.0s	1.5s	0.3s	0.8s

Table 9. Benefit of CG to **GridGraph** [43]: Average % reduction in the # of iterations requiring disk IO.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	23.5%	96.4%	44.4%	97.1%	95.6%	0%
TT	29.3%	94.8%	33.3%	94.1%	93.1%	42.0%
TTW	36.7%	94.7%	36.1%	94.5%	93.8%	0%
PK	27.5%	96.5%	47.0%	96.8%	92.4%	28.6%

For GridGraph, we specified 4×4 grid for partitioning the graphs and 8 GB of available memory which is less than all graph sizes (except PK). Since same memory configuration was used for all graphs, larger graphs experienced more IO and hence greater benefits from use of CGs. Figure 7 shows speedups observed for larger graphs are greater than for smaller ones. Note, speedups for FR>TT>TTW>PK due to higher disk IO savings. Fewer iterations in second phase are shown in percentage reduction terms in Table 9. The speedups for queries with high precision (SSNP, Viterbi, SSWP, REACH) speedups range from 13.62× to 1.35×. The speedups for SSSP and WCC are modest because the number of iterations in second phase is closer to number of iterations performed by the baseline GridGraph. Note that for WCC on FR there is no change in the number of iterations yet there is no slow down because fetches of more blocks of edges can be skipped in the second phase. For SSSP on PK there is a slight slow down because cost of first phase offsets the benefits to second. Finally, we observe that speedups achieved when AGs are used are relatively small ranging from 1.58× to a slowdown of 0.57×. We also note that bootstrapping initial result from CG is superior to AG. Note that higher speedups for AGs in [42] are due to additional optimizations besides "bootstrapping an initial result" from AG.

Ligra [34] runs on a server where the graph fits in memory and thus gains from CGs can be expected from reduced computation, and enhanced locality in the caches due to the smaller CG. For weighted graphs, as shown in Figure 8, the



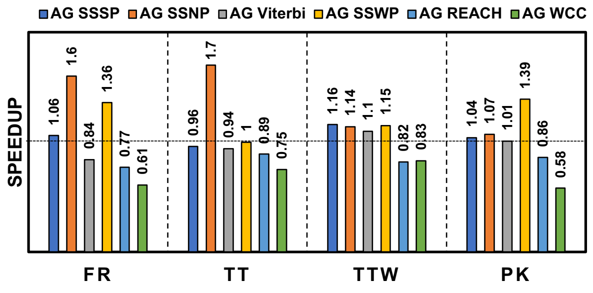


Figure 8. Speedups Over **Ligra** Due To Bootstrapping Initial Result from CG and AG.

Table 10. Average Execution Times in Seconds for Core Graph based 2Phase **Ligra** [34] across 10 Queries.

G	Specialized CGs				General CG	
	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	926.6s	358.1s	677.5s	405.1s	59.6s	377.7s
TT	137.7s	151.8s	186.4s	84.1s	28.3s	102.5s
TTW	235.4s	111.1s	130.5s	98.6s	25.2s	425.7s
PK	3.6s	1.8s	2.2s	2.4s	0.5s	1.2s

2Phase approach delivers speedups of 4.42–2.71× for SSWP queries (with highest first phase precision) and for SSSP queries (with lowest first phase precision) speedups of 1.44–1.08× were observed. For REACH speedups are even higher. Average speedups across all queries are higher for larger graphs (e.g., FR) and least for the smallest graph (PK). Note that to save memory needed for CG and FG, while preserving efficiency, the edge lists can be organized to separate critical and non-critical edges so that latter can be easily skipped during the first phase. We observe that in comparison to CGs, AGs deliver significantly lower speedups over Ligra: highest speedup of 1.70× for AGs vs. 9.31× for CGs. As we can see, AGs frequently result in slowdowns over Ligra. Finally, Table 11 shows that significant reduction in computation (dynamic edges processed) causes the Ligra performance to improve due to CGs.

The data presented thus far did not make use of the triangle inequality optimization thus requiring no major changes to existing systems. We added the optimization to Ligra and

Table 11. Benefit of CG to **Ligra** [34]: Average % Reduction in Edges Processed (EDGES-RED).

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	10.2%	26.1%	56.0%	50.4%	94.8%	40.9%
TT	46.2%	29.6%	36.4%	19.0%	93.1%	42.5%
TTW	52.5%	35.2%	51.9%	39.7%	92.1%	41.0%
PK	52.7%	39.1%	75.0%	44.3%	88.2%	36.8%

Table 12. Impact of Triangle Inequality on **Ligra** Speedups.

G		SSNP	Viterbi	SSWP
FR	SPEEDUP	4.24×	4.40×	7.30×
	EDGES-RED	70.95%	78.71%	93.23%
TT	SPEEDUP	6.06×	4.52×	6.01×
	EDGES-RED	89.95%	80.48%	88.80%
TTW	SPEEDUP	2.86×	2.78×	3.20×
	EDGES-RED	84.29%	80.75%	83.40%
PK	SPEEDUP	1.79×	1.83×	1.87×
	EDGES-RED	85.67%	86.71%	83.72%

reevaluated performance for the two largest input graphs, FR and TT. We consider the three algorithms identified in [14] for which the triangle inequality is the most effective. The new speedups shown in Table 12 for SSWP, Viterbi, and SSNP are substantial improvements over prior speedups for the two largest graphs.

3.3 Results for R-MAT Graphs

In addition to real data sets, we also used three generated R-MAT [8] graphs shown in Table 13(a).

- RMAT1 uses the same (a,b,c,d) parameters as used by Graph500 [26] and randomly generated edge weights with uniform distribution between 0 and 1 of single precision floats.
- RMAT2 is more dense, more locally connected, and with fewer long-range connections than RMAT1. This leads to smaller CGs than for RMAT1.
- RMAT3 is less dense, more globally connected, and with more long-range connections than RMAT1. This leads to larger CGs than for RMAT1.

Because these three R-MAT graphs are large (larger than FR data set), we used PaRMAT [15], a multi-threaded R-MAT graph generator, to generate them. Table 13 (b) and (c) show the small CG size and high precision of query results obtained from CG respectively. We observe that CGs sizes for R-MAT graphs are small, in fact even smaller than those for graphs considered earlier. The CGs for R-MAT graphs also deliver high precision ranging from 91.4% to 99.9%. Table 14 shows the speedups we obtained for Subway, Ligra, and GridGraph.

Table 13. R-MAT Graphs with 2.72 billion edges and 71.8 million vertices.

(a) Parameters and In-Memory Graph Size.

G	Parameters (a, b, c, d)	Size (GB)
RMAT1	**(0.57, 0.19, 0.19, 0.05)**	**22.0**
RMAT2	**(0.67, 0.14, 0.14, 0.05)**	**22.0**
RMAT3	**(0.47, 0.24, 0.24, 0.05)**	**22.0**

(b) % Edges in CGs.

G	SSSP	SSNP	Viterbi	SSWP	REACH
RMAT1	2.78%	2.70%	12.61%	2.70%	3.95%
RMAT2	1.68%	1.65%	7.82%	1.65%	3.17%
RMAT3	3.05%	2.98%	21.29%	2.89%	5.17%

(c) Precision of Queries Results.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
RMAT1	96.5%	99.9%	95.3%	99.9%	99.9%	99.9%
RMAT2	97.8%	99.9%	91.4%	99.9%	99.9%	99.9%
RMAT3	95.2%	99.9%	99.4%	99.9%	99.9%	99.9%

Table 14. Speedups for R-MAT graphs.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
	Subway					
RMAT1	2.32×	3.94×	1.02×	3.07×	2.81×	2.74×
RMAT2	2.51×	3.99×	0.97×	3.51×	2.28×	2.31×
RMAT3	2.42×	2.97×	0.89×	3.55×	4.46×	3.83×
	Ligra					
RMAT1	1.27×	1.47×	0.80×	3.52×	15.5×	1.50×
RMAT2	1.64×	1.30×	0.96×	2.76×	12.2×	2.32×
RMAT3	1.25×	1.35×	0.77×	1.98×	15.6×	1.61×
	GridGraph					
RMAT1	1.41×	5.24×	1.00×	5.22×	17.2×	2.55×
RMAT2	1.40×	4.72×	0.95×	5.17×	20.7×	1.90×
RMAT3	1.31×	3.81×	0.97×	4.55×	12.4×	1.62×

As we can see for these graphs with different characteristics, we also observed significant speedups. The only exception is Viterbi algorithm for which, due to lower precision and/or larger CG sizes, the costs of using core graphs often exceeds the benefits of using them.

3.4 Abstraction Graphs & Sampled Graphs vs. CGs

We also studied the precision of Abstraction Graph (AG) constructed according to the algorithm in [42]. The algorithm orders the edges according to increasing edge weights. First, pass over the edges adds those edges to the AG that connect two weakly connected components. Next pass includes additional edges till upper limit on number of allowed edges is reached – once again preference is given to lower weight edges. For fair comparison, we constructed AGs with equal number of edges as corresponding CGs and then compared their precision. The precision of AGs is given in Table 15. As we can see, the precision of AGs ranges from 6.1% to 69.9% while the precision of CGs ranges from 94.5% to 99.9%. We also doubled the size of AGs (relative to CGs) to see study how the precision improves. However, as shown by the rows labeled 2AG in Table 15, though precision improves to 6.2% to 83.9%, it is still far lower than that of CGs. The reason for high precision of CGs is preservation of key characteristics: power-law degree distribution and relative vertex degrees.

Table 15. Precision of AGs of sizes: (AG) equal to CG; (2AG) double of CG. % Vertices with Precise Results for 10 Queries.

G		SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	AG-P	22.3%	52.4%	35.9%	52.7%	25.5%	9.4%
	2AG-P	36.2%	63.9%	62.7%	63.9%	44.6%	58.1%
TT	AG-P	34.4%	43.2%	27.9%	43.2%	26.9%	6.1%
	2AG-P	50.6%	61.6%	55.0%	63.4%	55.7%	6.2%
TTW	AG-P	29.0%	60.3%	23.7%	55.4%	43.5%	54.0%
	2AG-P	46.0%	77.9%	46.9%	77.7%	53.1%	67.8%
PK	AG-P	46.8%	69.9%	14.4%	71.8%	49.5%	59.6%
	2AG-P	73.5%	83.9%	44.7%	85.6%	62.3%	75.5%

Table 16. Precision of SGs of sizes: (SG) equal to CG; (2SG) double of CG. % Vertices with Precise Results for 10 Queries.

G		SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	SG-P	9.8%	15.2%	11.8%	12.7%	35.2%	38.1%
	2SG-P	11.2%	19.7%	15.1%	17.2%	39.7%	42.5%
TT	SG-P	8.7%	10.5%	15.5%	6.3%	41.8%	33.5%
	2SG-P	11.8%	14.1%	17.1%	10.1%	49.2%	35.3%
TTW	SG-P	10.4%	11.2%	18.8%	15.9%	34.8%	47.9%
	2SG-P	12.8%	14.8%	21.5%	18.1%	38.2%	51.7%
PK	SG-P	11.1%	14.9%	14.6%	17.8%	30.6%	48.5%
	2SG-P	15.0%	17.6%	19.5%	21.5%	32.7%	58.4%

Sampling techniques have been developed to scale down graph size [21, 29, 41]. We generated Sampled Graphs (SGs) using random walks [41] and used them in place of CGs for two phase evaluation of queries. The precision data of this approach is given in Table 16. We observe that overall the precision of SGs is even lower than AGs. This is because sampling does not guarantee creation of well-connected graphs.

In contrast, CGs provide highly precise results for the following reasons. First, since CGs are built from query results, they include paths formed by non-zero centrality edges giving us good connectivity. Second, as shown by a representative plot in Figure 9, the degree distributions of FG and CG are similarly power law. Third, as shown in Table 17, although the degrees of high degree vertices in CG are reduced, their relative degrees remain unchanged (e.g., the top 1000 vertices in CG and full graph are exactly the same).

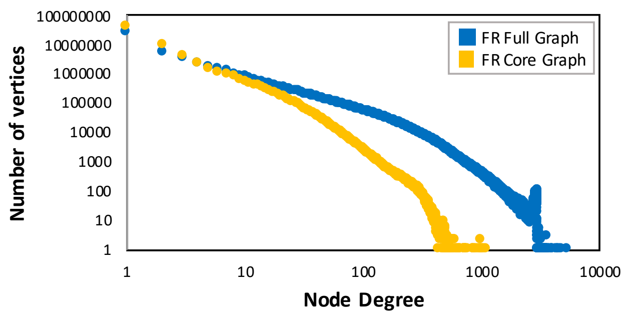


Figure 9. Power Law Degree Distribution of Full Graph vs. Core Graph: FR for SSSP.

Table 17. Degree of Overlap in Sets of Topmost 100,000 Highest Degree Vertices between FGs and CGs for SSSP.

G	Common High Degree Vertices		
	Top 1,000	Top 10,000	Top 100,000
FR	1,000	9,997	99,931
TT	1,000	10,000	99,997
TTW	1,000	10,000	99,988
PK	1,000	10,000	98,988

4 Related Work

A Proxy Graph used for evaluating all future queries. We discuss relevant works that derive a smaller graph for a large graph to speedup evaluation of queries. *Input reduction* [18] employs property preserving graph transformations to reduce graph size and then uses 2Phase processing. However, transformations eliminate vertices and graph size reductions are limited. Smallest reduced graph had around 50% of the edges and it can only be used to evaluate queries for subset of vertices in the full graph. Abstraction Graph [42] delivers a small proxy graph; however, it lacks precision as shown in Table 15. *In contrast, core graphs are much smaller (5.42–10.45% for FR) and yet produce accurate results for over 94% vertices, and leaving less work for the second phase. Graph Sampling* scales down the size of a graph [21, 29, 41] while preserving global graph characteristics. However, our experimental results in Table 16 show that the ability to solve arbitrary queries with high precision is lost as the sampled graph may not be well-connected thus eliminating paths between vertices.

Query specific pruning for Point-to-Point queries. Our work focuses on evaluating vertex queries that originate at a source vertex and then compute property values for all other vertices that are reachable from it. Another class of queries, point-to-point queries, compute a property value between a source and destination vertex pair. As the first step graph pruning is performed and then the query is evaluated on the *pruned graph* [39, 40]. Unlike Core Graph, pruned graph must be recomputed for each new query. Due to limited scope of a point-to-point query, pruning parts of the graph that do not fall on paths from source to destination significantly reduces graph size. However, since our work is for point-to-all queries, pruning would only reduce the graph size minimally. Next we provide comparison with two specific point-to-point query algorithms.

Query-by-Sketch (Qbs) [39] is a three-phase (offline labelling, query specific online sketching, and searching) algorithm for **finding shortest path between two vertices** (i.e., shortest path point-to-point query). Though core graph and Qbs speedup query evaluation over large graphs, Qbs has major limitations. First, a sketch is query specific and thus must be computed online for each query while the core graph is found once and used for all queries. Second, core graph is *general* as it solves many kinds of queries as opposed Qbs that is for only shortest path query. Third, we evaluate demanding queries that compute property values from *one source vertex to all other reachable vertices* while Qbs evaluates a single point-to-point query. For the queries we evaluate, a sketch is expected to be very large fraction of the graph. Finally, not only is online sketching expensive, if the sketch produced is large in size, then computation of shortest path takes a long time. For TTW labelling and sketching takes 1,345 seconds and the sketch is large (0.76GB). *However, core graph construction is relatively efficient and done once.*

Pruning and Prediction (PnP) [40] is another method for evaluating point-to-point queries. This algorithm employs bidirectional BFS originating from source (forward) and destination (backward) to first prune the graph for a given query and then perform remainder of the query evaluation. The pruning is query specific like the query sketch used by Qbs.

Query specific transient graphs generated on-the-fly. Transient graphs are generated to minimize data transfer cost multiple times during the evaluation of a query. There are two contexts in which such work has been done: works considering graphs that do not fit in GPU memory; and out-of-core systems for graphs that do not fit in the memory of a single machine. Both these approaches can benefit from Core Graphs to reduce data transfers – between host and GPU-memory vs. disk and machine memory.

A number of approaches have been developed to reduce data transfer in context of a GPU [12, 17, 22, 32, 33]. Among them Subway is the most promising – it on-the-fly generates and loads *transient active subgraphs* covering the frontier from one iteration to next. Across iterations, the reachable graph is loaded at least once. *In contrast, a core graph is loaded in its entirety once and computes precise results for over 94% of vertices without requiring graph transfers.*

On shared-memory machines when graphs cannot fit in memory, out-of-core partition-based processing is used [20, 31, 37, 42, 43]. Partitions are loaded from disk one at a time and processed. Typically the disk IO represents 70% of the runtime cost [37]. To reduce disk IO, [37] maximizes the work performed on one partition before loading the next partition. Wonderland [42] organizes edges across partitions according to their weights so fewer passes, and faster convergence, can be obtained. Nevertheless, the cost of disk IO is high. *Out-of-core systems benefit from our approach since first phase loads the core graph and computes precise results for over 94% of vertices without additional IO.*

5 Concluding Remarks

We identified *core graph*s with 10.7% of edges on average that rapidly yield precise results for 94.5–99.9% of vertices. An optimized second pass efficiently computes the precise results for rest of the vertices. The generality of the CG based approach allows it to be applied across existing systems without requiring any major modifications to them. We demonstrated this by enhancing three different systems. Significant performance improvements for these systems were observed – up to 4.48× in Subway [32], up to 13.62× in GridGraph [43], and up to 4.97× in Ligra [34]. Applying the triangle inequality optimization gave additional speedups. While these results were for real data sets corresponding to power law graphs, we also observed performance improvements for generated R-MAT graphs with varying charateristics.

Acknowledgments

We thank the reviewers for their feedback. This work is supported in part by National Science Foundation Grants CNS-1955650, CNS-2053383, CCF-2028714, CCF-2002554, and CCF-2226448 to the University of California, Riverside.

References

[1] Friendster data set. In *http://konect.cc/networks/friendster/*.

[2] Snap: Stanford network analysis platform. In *https://snap.stanford.edu/snap/*.

[3] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Vancouver, BC, Canada, March 25–29, 2023*, pages 133–145. ACM, 2023.

[4] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing (HOPC'23), Orlando, FL, USA, June 16, 2023*, pages 1–2, 2023.

[5] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graph analytics on evolving data (abstract). In *arXiv preprint arXiv:2308.14834*, 2023.

[6] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 235–248, 2017.

[7] M. Cha, H. Haddadi, Fabrício Benevenuto, and K. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *ICWSM*, 2010.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM Data Mining*, 2004.

[9] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Mega evolving graph accelerator. In *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, 2023.

[10] Jennifer Golbeck. *Analyzing the Social Web*. Morgan Kaufmann, 2013.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[12] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Proceesdings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, PACT '17, pages 233–245, 2017.

[13] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. Multigraph: Efficient graph processing on gpus. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, PACT '17, pages 27–40, 2017.

[14] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Tripoline: generalized incremental graph processing via graph triangle inequality. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 17–32. ACM, 2021.

[15] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[16] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, pages 239–252. ACM, 2014.

[17] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 447–461, 2016.

[18] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*, pages 245–257. ACM, 2016.

[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and S. Moon. What is twitter, a social network or a news media? In *WWW '10*, 2010.

[20] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46. USENIX Association, 2012.

[21] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 631–636, New York, NY, USA, 2006. Association for Computing Machinery.

[22] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 49–63, 2019.

[23] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[24] LongJason Lu and Minlu Zhang. *Edge Betweenness Centrality. In Encyclopedia of Systems Biology*, pages 647–648. Springer New York, New York, NY, 2013.

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[26] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[27] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, 2013.

[28] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[29] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. C-SAW: a framework for graph sampling and random walk on gpus. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 56. IEEE/ACM, 2020.

[30] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1091–1105, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[32] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth EuroSys Conference*, EuroSys '20, pages 12:1–12:16, 2020.

[33] Dipanjan Sengupta, Kapil Agarwal, Shuaiwen Leon Song, and Karsten Schwan. Graphreduce: Large-scale graph analytics on accelerator-based HPC systems. In *IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '15, pages 604–609, 2015.

[34] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[35] L. Takac. Data analysis in public social networks. 2012.

[36] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 223–236, 2017.

[37] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In Ajay Gulati and Hakim Weatherspoon, editors, *USENIX Annual Technical Conference (USENIX ATC) 2016, Denver, CO, USA, June 22-24, 2016*, pages 507–522. USENIX Association, 2016.

[38] Yangzihao Wang, Yuechao Pan, Andrew A. Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing*, 4(1):3:1–3:49, 2017.

[39] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. Query-by-sketch: Scaling shortest path graph queries on very large networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1946–1958. ACM, 2021.

[40] Chengshuo Xu, Keval Vora, and Rajiv Gupta. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 587–600. ACM, 2019.

[41] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. Knightking: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 524–537, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 608–621. ACM, 2018.

[43] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference (USENIX ATC), July 8-10, Santa Clara, CA, USA*, pages 375–386, 2015.