Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing

Xizhe Yin
xyin014@ucr.edu
University of California, Riverside
USA

Zhijia Zhao
zhijia@cs.ucr.edu
University of California, Riverside
USA

Rajiv Gupta
gupta@cs.ucr.edu
University of California, Riverside
USA

ABSTRACT

In concurrent graph processing, different queries are evaluated on the same graph simultaneously, sharing the graph accesses via the memory hierarchy. However, different queries may traverse the graph differently, especially for those starting from different source vertices. When these graph traversals are "misaligned", the benefits of graph access sharing can be seriously compromised. As more concurrent queries are added to the evaluation batch, the issue tends to become even worse.

To address the above issue, this work introduces `Glign`, a runtime system that automatically aligns the graph traversals for concurrent queries. `Glign` introduces three levels of graph traversal alignment for iterative evaluation of concurrent queries. First, it synchronizes the accesses of different queries to the active parts of the graph within each iteration of the evaluation—*intra-iteration alignment*. On top of that, `Glign` leverages a key insight regarding the "heavy iterations" in query evaluation to achieve *inter-iteration alignment* and *alignment-aware batching*. The former aligns the iterations of different queries to increase the graph access sharing, while the latter tries to group queries of better graph access sharing into the same evaluation batch. Together, these alignment techniques can substantially boost the data locality of concurrent query evaluation. Based on our experiments, `Glign` outperforms the state-of-the-art concurrent graph processing systems `Krill` and `GraphM` by 3.6× and 4.7× on average, respectively.

CCS CONCEPTS

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

KEYWORDS

concurrent graph processing, data locality, graph system, iterative graph algorithm, graph traversal

ACM Reference Format:
Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2023. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3567955.3567963

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9915-9/23/03.
https://doi.org/10.1145/3567955.3567963

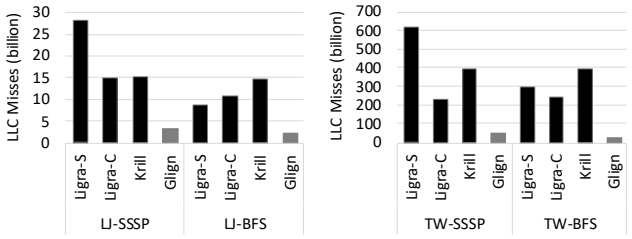

Figure 1: Last-Level Cache Misses (64 concurrent queries on LiveJournal [3] and Twitter [12], measured by `perf` profiler).

1 INTRODUCTION

Although last decade witnessed significant advances in developing efficient graph processing systems, supports for concurrent query evaluation remain underexplored. Most existing graph processing systems are designed to process one analytical query each time, such as a single-source shortest path (SSSP) query. On the other hand, as the demands of graph analytics grow, so do the needs for concurrent evaluation of graph queries [34, 38, 42]. A prior study on social network application shows that most graph query jobs are executed concurrently [34]. To fill this gap, several concurrent graph processing systems [6, 20, 34, 38, 42] have been proposed in recent years, including `Seraph` [34] for distributed platforms, `CGraph` [38] and `GraphM` [42] with supports for out-of-core processing, and `Congra` [20] and `Krill` [6] which focus on in-memory evaluation of a batch of concurrent graph queries.

Opportunities and Challenges. By evaluating multiple queries simultaneously on a graph, concurrent graph processing enables graph access sharing across queries via the memory hierarchy, that is, the graph data fetched to the cache(s) by one query may be used directly by other queries. Intuitively, such sharing may reduce the total cache misses, benefiting the overall performance. However, this work finds that the actual cache miss reduction brought by concurrent graph query evaluation could be quite limited.

Figure 1 reports the last-level cache (LLC) misses of evaluating 64 concurrent queries on two graphs using some representative graph systems. As a baseline, `Ligra-S` evaluates the queries one by one using `Ligra` [28], a well-known in-memory graph processing framework that evaluates each query in parallel. In comparison, `Ligra-C` evaluates all 64 queries simultaneously using an extended `Ligra` with basic concurrency supports (see Section 4); `Krill` is a state-of-the-art concurrent graph processing system just released recently [6]. As the results show, even with a concurrency degree of 64 queries, the cache misses of `Ligra-C` and `Krill` are reduced by a

limited fraction comparing to Ligra-S and sometimes, their cache misses may even exceed that of the baseline (LJ-BFS and TW-BFS).

A primary reason causing the above unfavorable results lies in *the potential "misalignment" of underlying graph traversals among concurrent queries*. Though the above 64 queries are of the same type, they may traverse the graph very differently due to their vertex-specific nature (i.e., starting from different source vertices). For queries of different types, their underlying graph traversals can be even more diverse. When the traversals are misaligned—visiting different parts of the graph for most of the processing time, the concurrent evaluation of queries will not benefit much from the shared memory accesses. Even worse, they may even "hurt" each other by competing for the caches.

Solution of This Work. To address the above issue, this work proposes a runtime system for in-memory graph processing on multi-core platforms [1], namely, Glign [2]. Glign can automatically align different graph traversals of concurrent queries to maximize the graph access sharing. As a result, it can significantly reduce the cache misses compared to other systems (see Figure 1). Glign primarily targets vertex-specific queries that employ iterative graph algorithms for evaluation, such as SSSP and BFS. In addition, to benefit the most from Glign, the vertex function of the iterative algorithms $f(v)$ needs to be monotonic, a common property shared by many vertex-centric graph query algorithms [11, 25, 31]. Next, we briefly introduce the key techniques behind Glign.

First, like most existing concurrent graph processing systems [6, 38, 42], Glign synchronizes the iterations of different queries during evaluation—the barriers used for iterative evaluation are shared across queries. This design allows Glign to treat the iterations as a logical timeline for aligning graph traversals. To distinguish them from the iterations in single-query graph processing, we refer to the iterations shared by queries as *global iterations*.

Based on the global iterations, Glign addresses the problem of graph traversal misalignment at three levels:

- *Intra-iteration alignment*. In each iteration of the evaluation, a query needs to access an active part of the graph (a.k.a. *frontier*). Intuitively, the active parts of different queries may overlap. If the overlapped parts are accessed around the same time, the evaluation will benefit from temporal locality.
- *Inter-iteration alignment*. For a given batch of queries, Glign allows their evaluation to start at *different* global iterations, thus making it possible to align the iterations across queries based on their graph access sharing.
- *Alignment-aware batching*. At the high level, considering all the concurrent queries available, which queries should be put into the same evaluation batch? Different batching strategies may yield different amounts of graph access sharing.

For intra-iteration alignment, existing designs [6, 32] require two levels of frontiers to achieve synchronized frontier traversal. Instead, Glign proposes *query-oblivious frontier*, a single-level frontier that deliberately ignores the frontier differences across queries. This is possible if the vertex function of the query is monotonic. On the other hand, this may evaluate extra vertices due to its inability to distinguish some inactive ones for certain queries. Overall, we found the benefits of reduced memory (from the use of a single-level frontier) easily outweighs the side effects of extra computations.

[1] Similar ideas could be applied to out-of-core and distributed processing scenarios.
[2] Pronounced as /gline/.

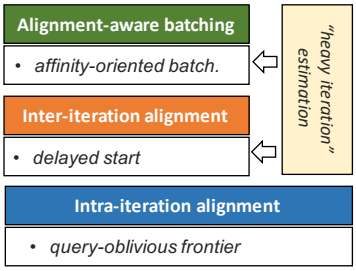

Figure 2: Three Levels of Alignments in Glign.

For inter-iteration alignment and alignment-aware batching, Glign leverages an important insight revealed in this work:

the "heavy iterations" of concurrent queries should be well aligned during the evaluation.

Here, "heavy iterations" refer to iterations that access a relatively larger portion of the graph (i.e., a large frontier). The insight is backed by two facts. First, heavy iterations often dominate the total processing cost of a query; Second, larger frontiers often expose more opportunities for intra-iteration alignments—a potentially larger overlapping among the frontiers of different queries.

The above insight reduces the two higher-level alignments into the problem of *alignment of heavy iterations*. To solve the latter, this work uses a simple yet effective heuristic to estimate the arrival time of heavy iterations. Based on the estimation, two scheduling techniques are proposed to improve the alignments:

- *Delayed start*. For a given batch of concurrent queries, this technique postpones the start of the evaluation of certain queries to later global iterations, based on the arrival time differences of their heavy iterations;
- *Affinity-oriented batching*. Considering all the concurrent queries received, it groups queries with closer arrival time of heavy iterations (affinity) to the same evaluation batch.

Figure 2 lists the above techniques. To confirm their effectiveness, this work evaluated Glign with commonly used graphs and query benchmarks, and compared it with two state-of-the-art concurrent graph systems: GraphM [42] and Krill [6]. The results show that the proposed alignment techniques can reduce the LLC misses by a significant ratio. They also show that Glign achieves on average 3.6× speedup over Krill and 4.7× speedup over GraphM.

In summary, this work makes a three-fold contribution:

- First, it reveals a key performance issue in concurrent graph processing—graph traversal misalignments, and categorizes it at three levels of the graph processing system.
- Second, it proposes a series of techniques to address the misalignments at each level: a new design of synchronized frontier traversal and two scheduling techniques based on the insight of heavy iterations.
- Finally, it integrates the above techniques in a system Glign and compares it with the state-of-the-art systems.

Next, we start with some background of this work.

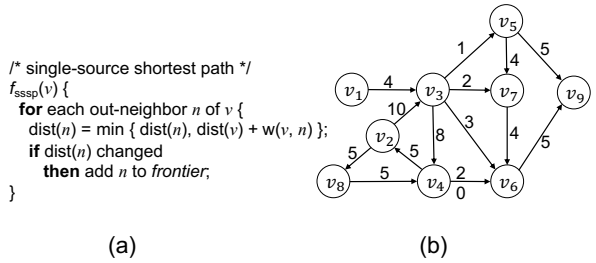

Figure 3: Example Vertex Function and Graph.

Table 1: Iterative Evaluation of sssp(v_1)

Iter#	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	Frontier
0	0	∞	∞	∞	∞	∞	∞	∞	∞	{v_1}
1	0	∞	4	∞	∞	∞	∞	∞	∞	{v_3}
2	0	∞	4	12	5	7	6	∞	∞	{v_4, v_5, v_6, v_7}
3	0	17	4	12	5	7	6	∞	10	{v_2, v_9}
4	0	17	4	12	5	7	6	22	10	{v_8}
5	0	17	4	12	5	7	6	22	10	{}

2 BACKGROUND

This section introduces the basics of vertex-centric graph processing and the idea of concurrent graph query evaluation.

2.1 Vertex-Centric Graph Processing

Following the idea of "thinking like a vertex" [18], vertex-centric graph processing emerges as the de facto model for programming graph applications. Many graph systems have been proposed in recent years based on this model, such as Pregel [15], GraphLab [14], Giraph [26], PowerGraph [9], and Ligra [28]. Under this model, a *vertex function* $f(v)$ needs to be specified by developers, which will be evaluated on every vertex in the graph or a selected subset of vertices (i.e., *frontier*) iteration by iteration, to compute certain vertex-specific properties, like the shortest distances from a source vertex to every other vertex. The iterations stop when the properties of all vertices stop changing (convergence) or some thresholds are met, following the bulk synchronous parallel (BSP) model [30].

Take single-source shortest path (SSSP) as an example. The goal is to find the shortest distance from a source vertex to every other vertex in the graph. Figure 3-(a) shows its vertex function $f_{sssp}(v)$, which updates the out-neighbors of vertex v based on its current value. Considering the graph in Figure 3-(b), the evaluation process of query sssp(v_1) is given in Table 1, including the vertex values and the frontier in each iteration.

Initially, the values of all vertices are set to ∞, except for the source vertex v_1, and only v_1 is in the frontier (i.e., activated). After applying the vertex function to v_1, the (only) out-neighbor of v_1, which is v_3, obtains a new value 4. As a result, v_3 becomes the new frontier. As the evaluation proceeds, the frontier is propagated through the graph, along with new vertex values being generated iteration by iteration, until the frontier becomes empty—the *fixed point*. The resulted vertex values are the answer to the query.

Note that the vertex function in Figure 3-(a) needs to access and update the out-neighbors of a vertex. This is known as the *push model*. Alternatively, the vertex function might also be designed to access and update the in-neighbors of a vertex, which is referred to as the *pull model*. In this work, we assume a push model is chosen.

2.2 Concurrent Evaluation of Graph Queries

Recently, several graph processing systems have been proposed to support concurrent graph query evaluation, such as `Seraph` [34], `CGraph` [38], `GraphM` [42], `Congra` [20], and `Krill` [6], covering distributed, in-memory, and out-of-core processing scenarios. As to the execution model, `Seraph`, `CGraph`, and `GraphM` all treat each concurrent query as a "job". In `Seraph`, the minimum execution unit is a task, consisting a bunch of vertices to be processed in a job, while different tasks are processed using a thread pool. In comparison, `CGraph` maps each concurrent job to a core by default, then balances the jobs across cores. `GraphM` is built on top of the out-of-core graph processing system `GridGraph` [43], where edges are partitioned into "blocks" and processed by worker threads. Finally, `Congra` [20] and `Krill` [6] are both built on top of `Ligra` [28], a state-of-the-art in-memory graph processing system. Under the hood, `Ligra` exploits the vertex-level parallelism where the vertex function is applied to the active vertices (in the frontier) in parallel, guided by a work stealing scheduler (from Cilk [4]). So even for a single query, the system can evaluate it in parallel with relatively balanced workload across CPU cores.

Table 2: Graph Access Sharing between Two Queries.

Iter#	Frontier(sssp(v_2))	Frontier(sssp(v_8))
0	{v_2}	{v_8}
1	{v_3, v_8}	{v_4}
2	{v_4, v_5, v_6, v_7}	{v_2, v_6}
3	{v_9}	{v_3, v_9}
4	{}	{v_5, v_6, v_7}
5	{}	{v_9}
6	{}	{}

A key potential benefit of concurrent graph query evaluation is the sharing of graph accesses via the memory hierarchy, which improves the overall data locality. Consider two queries, sssp(v_2) and sssp(v_8), to the graph in Figure 3-(b). In fact, both queries need to access the out-neighbors of vertices v_2 - v_9 during the evaluation, as indictated by their frontiers in Table 2. If the graph data fetched by one query still resides in the shared cache when the other query tries to access it (i.e., temporal locality), the overall cache misses could be dramatically reduced. However, for many real-world graphs, their sizes are well beyond the cache capacity. In order to benefit from this temporal locality, the graph traversals should be roughly *aligned*—visiting the same vertices (and their out-neighbors) around the same time.

In fact, as reported earlier in Figure 1, the underlying graph traversals on real-world graphs could be largely misaligned in the existing concurrent graph processing systems, limiting the benefits of shared graph accesses. In the following, we will present a solution to addressing the graph traversal misalignment issue—`Glign`.

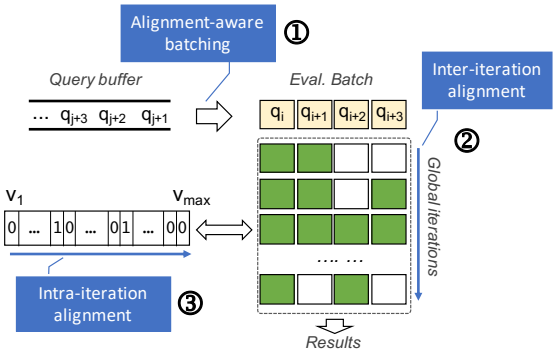

Figure 4: Overview of `Glign`.

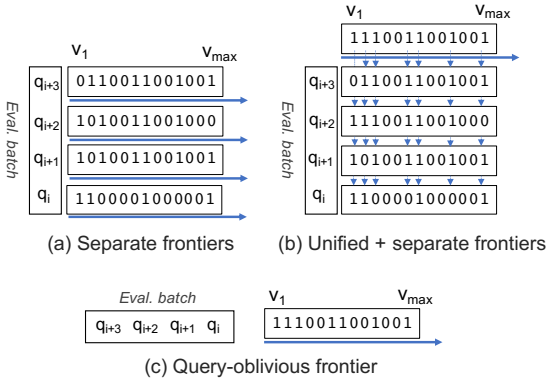

Figure 5: Different Designs of Frontier Traversal.

3 Glign

Figure 4 illustrates the high-level design of `Glign`. First, assume a buffer consists of the concurrent queries to be evaluated, `Glign` uses an *alignment-aware batching* strategy to group queries with higher potential of a large amount of graph access sharing to the same evaluation batch (see ①). After a batch is formed, `Glign` performs *inter-iteration alignment* across (global) iterations. In particular, it first estimates the "heavy iterations" for each query in the batch, then *delays* some of them to make the "heavy iterations" of different queries aligned (see ②). Finally, within each global iteration, `Glign` ensures that the frontiers of different queries are traversed in a synchronous manner so that the shared graph accesses (dictated by the overlapping of frontiers) are accessed in a fully coalesced way (see ③). Next, we will present each of these key techniques in detail. Due to their dependences, we will introduce them in reverse order with respect to the number labels in Figure 4.

3.1 Global Iterations

First, we introduce the concept of *global iterations*, which serve as the basis for some of the proposed alignments. Given a batch of iterative graph queries, there are two ways to evaluate them:

- *Synchronous evaluation* evaluates queries in the batch in the same pace with respect to iterations (see Figure 4). This is ensured by a series of global barriers that are shared across queries in the batch. Most existing concurrent graph systems (`CGraph` [38], `GraphM` [42] and `Krill` [6]) follow this scheme.
- *Asynchronous evaluation* evaluates each query in the batch independently, regardless of the evaluation pace of other queries, that is, the iterations of evaluating different queries may be interleaved arbitrarily. `Congra` [20] uses this scheme.

Clearly, the asynchronous design has no control over the graph traversals, so the traversals may or may not align well depending on their interleaving in a specific evaluation. For this reason, `Glign` follows the synchronous evaluation. To distinguish the iterations in the synchronous batch evaluation from those in single-query evaluation, we refer to the former as *global iterations*.

3.2 Intra-Iteration Alignment

A commonly used design for evaluating concurrent graph queries is to keep a frontier for each query q_i in the evaluation batch $\mathcal{B}$. In a global iteration, the frontiers of different queries are traversed independently, as shown in Figure 5-(a). The frontier is designed as a boolean array `frontier[]`, where the i-th element shows the activeness of vertex v_i. If `frontier[i]=1`, the vertex function $f(v)$ needs to be evaluated on v_i, including accessing the out-neighbors of v_i. In other words, the frontier traversal defines how graph data is accessed in a global iteration. When these frontiers of different queries are traversed independently, there is no guarantee that the commonly used graph data are accessed around the same time. As a result, the data locality could become sub-optimal.

To ensure that different frontiers are traversed in a synchronized manner, some recent works (`Krill` [6] and `SimGQ` [32]) propose to add an extra frontier, called *unified frontier*, defined as follows:

$$Frontier_{union} = \bigvee_{q_i \in \mathcal{B}} Frontier_{q_i} \quad (1)$$

where $Frontier_{q_i}$ (a boolean array) is the frontier for evaluating query q_i and $\bigvee$ is the logical OR operator. This means that as long as vertex v_i is active for one query in the batch, $Frontier_{union}(i) = 1$. To synchronize the frontier traversals, we can simply traverse the unified frontier: if its value for vertex v_i is "1", we further check each individual frontier $Frontier_{q_i}$ to find out the specific queries for which v_i needs to be evaluated (see Figure 5-(b)).

The above design ensures that the shared accesses to an active vertex and its out-neighbors are perfectly aligned across queries. However, there are some caveats associated with this design. First, it increases the memory cost with an extra labeling array $Frontier_{union}$; Second, it needs to check the frontiers at two levels. Overall, our evaluation reports limited performance benefits (see Section 4).

To avoid the above caveats, this work proposes an alternative design to the synchronized frontier traversal, called *query-oblivious frontier*. This new design explores an interesting tradeoff between computations and memory accesses, which to our best knowledge, has not yet been discussed before by any prior work.

Query-Oblivious Frontier. Our key insight is *to deliberately ignore the differences among the frontiers of queries in the evaluation batch*, that is, when a vertex function $f(v)$ is invoked, it is applied for all queries in $\mathcal{B}$. This eliminates the need of second-level frontiers

($Frontier_{q_i}$, q_i in $\mathcal{B}$) used in the prior design. Figure 5-(c) illustrates this idea with a single frontier $Frontier_{union}$.

However, the above design immediately raises two concerns:

- *Correctness.* Does the evaluation based on a single unified frontier ($Frontier_{union}$) always produce the same results as the one using two levels of (or seperate) frontiers?
- *Efficiency.* A vertex v would be evaluated for all queries in the batch, as long as it is in the frontier of one query. This introduces extra unnecessary computations.

First, for correctness, we have established a theorem for safely adopting query-oblivious frontier for a range of iterative queries based on the *monotonicity* property of their vertex functions.

Definition 3.1. In vertex-centric programming, a vertex function $f(\cdot)$ is **monotonic** iff. it always changes the values of vertices monotonically (always increasing or decreasing) over iterations.

In fact, the monotonicity property has been widely exploited by multiple existing graph systems for better efficiency [14, 25] and it serves as the basis for incremental query evaluation [11, 31].

Theorem 3.2. *Evaluating a query batch using query-oblivious frontier yields the same vertex values as the evaluation using separate frontiers iff. the vertex function is monotonic.*

Proof. Without loss of generality, assume the evaluation batch consists of two queries q_1 and q_2, and the evaluation is in global iteration i. Consider an arbitrary vertex v_j. Assume v_j is inactive according to q_1's frontier, but active based on q_2's frontier. Thus, v_j is marked as an active one in the unified frontier, as illustrated in Figure 6. Next, we discuss the impacts of an extra evaluation of v_j with q_1's vertex function, that is, $f_{q_1}(v_j)$. There are two basic cases. In the first case, at the time of evaluating $f_{q_1}(v_j)$, the value of v_j has not been updated by any of its in-neighbors. As its value remains the same as it was in the prior iteration $i-1$, this evaluation will not change the value of any of its out-neighbors (like v_t). In the second case, at the time of evaluating $f_{q_1}(v_j)$, the value of v_j has been updated by at least one of its in-neighbors in the current iteration i. In this case, the evaluation may update the value(s) of some out-neighbor(s) of v_j, causing some side-effects. However, note that even if separate frontiers are used, v_j would be marked as an active vertex and evaluated in the next iteration $i+1$. That is, using query-oblivious frontier might lead to some earlier evaluation of certain vertices that are supposed to be evaluated in the next iteration—a form of asynchronous evaluation. One sufficient condition for the correctness of asynchronous query evaluation is that the query evaluation should be monotonic. □

Second, as to the efficiency concern, will the extra evaluation of inactive vertices slow down the overall processing? Interestingly, our evaluation (see Section 4) shows that using query-oblivious frontier can substantially improve the overall performance despite the extra evaluation of inactive vertices. This is due to fact that query-oblivious frontier skips the maintenance and accesses to the separate frontiers all together, dramatically reducing the memory footprint of concurrent query evaluation (see Section 4).

So far, we have introduced the intra-iteration alignment which addresses the potential misalignments among different frontier traversals in a global iteration. Next, we will shift the focus to more coarse-grained misalignments, among the iterations of different queries and during the formation of query batches.

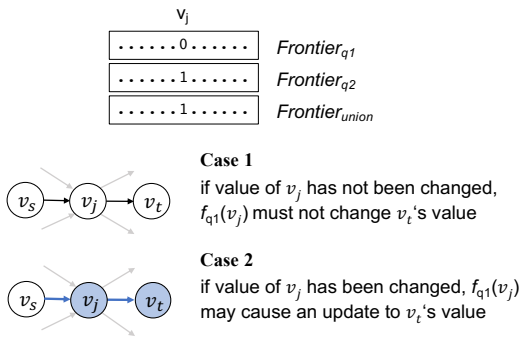

Figure 6: Correctness of Using Query-Oblivious Frontier.

3.3 Inter-Iteration Alignment

We first use a simple example to motivate the alignment problem, then formalize it and present a heuristic-based solution.

Motivation. In general, the evaluation of a graph query may access different parts of the graph in different iterations, thus the amount of graph sharing may vary depending on the interleaving of the (local) iterations of different queries. Revisit the examples in Table 2 (Section 2) and assume the two sssp queries are evaluated in the same batch, then compare their frontier overlapping per iteration with those in Table 3 where a different alignment between the (local) iterations of the two queries is used: sssp(v_2) starts two iterations later than sssp(v_8). From the comparison, we can find that the latter alignment exposes more overlapped active vertices than the former (6 v.s. 2). As a result, when these active vertices are evaluated and their (out-)neighbors are accessed, the latter alignment will yield more shared graph accesses.

Table 3: A Better Alignment of Iterations.

Iter#	Frontier(sssp(v_2))	Frontier(sssp(v_8))
0	–	$\{v_8\}$
1	–	$\{v_4\}$
2	$\{v_2\}$	$\{v_2, v_6\}$
3	$\{v_3, v_8\}$	$\{v_3, v_9\}$
4	$\{v_4, v_5, v_6, v_7\}$	$\{v_5, v_6, v_7\}$
5	$\{v_9\}$	$\{v_9\}$
6	{}	{}

Next, we formalize the above inter-iteration alignment problem.

Problem Formalization. First, we define the alignment vector I for a given batch of queries $\mathcal{B}$ as follows:

Definition 3.3. Given a batch $\mathcal{B}$ of n queries $[q_1, q_2, ..., q_n]$, its ***alignment vector*** I is a vector of $[a_1, a_2, ..., a_n]$, where a_i is the global iteration number from which the evaluation of q_i is started.

Considering the batch $\mathcal{B}$=[sssp(v_2), sssp(v_8)], Table 3 shows an alignment where the alignment vector I = [2, 0].

Next, we introduce the concept of *affinity* to quantify the amount of graph access sharing, which is defined as follows:

Definition 3.4. Given a query batch $\mathcal{B}$ and an alignment vector I, the **affinity** of this evaluation is defined by the following equation:

$$Affinity(\mathcal{B}, I) = 1 - \frac{\sum_{j=0}^{K} |Frontier^{j}_{union}|}{\sum_{j=0}^{K} \sum_{q_i \in \mathcal{B}} |Frontier^{j}_{q_i}|} \quad (2)$$

where $Frontier^{j}_{union}$ and $Frontier^{j}_{q_i}$ are the unified frontier and the separate frontier for query q_i, respectively, at iteration j, and K is the total number of global iterations for evaluating this batch.

Again, consider the examples in Table 3, we have

$$Frontier^{0}_{union} = \{v_8\}, Frontier^{1}_{union} = \{v_4\},$$
$$Frontier^{2}_{union} = \{v_2, v_6\}, Frontier^{3}_{union} = \{v_3, v_8, v_9\},$$
$$Frontier^{4}_{union} = \{v_4, v_5, v_6, v_7\}, Frontier^{5}_{union} = \{v_9\}.$$

Thus, $Affinity(\mathcal{B}, I) = 1 - (1 + 1 + 2 + 3 + 4 + 1)/(8 + 10) = 1/3$. By contrast, the affinity for the alignment in Table 2 $Affinity(\mathcal{B}, I') = 1 - (2 + 3 + 5 + 2 + 3 + 1)/(8 + 10) = 1/9$. Obviously, the former achieves a significantly higher affinity.

The best affinity occurs when all separate frontiers are perfectly overlapped, while the worst affinity happens when no separate frontiers overlap at all through all the iterations. Note that the affinity may be negative in certain cases, due to the asynchronous evaluation as detailed in the proof of Theorem 3.2.

The above definition of affinity is based on the ratio of active vertices. Alternatively, one can also define affinity based on the ratio of active edges (the outgoing edges of active vertices), which is supposed to be more precise. However, our evaluation shows minimal differences between the two definitions in practice.

Now we formalize the inter-iteration alignment as follows:

$$\max_{\forall I} \ Affinity(\mathcal{B}, I) \quad (3)$$

That is, given an evaluation batch $\mathcal{B}$, the problem is to find out the best alignment vector I that maximizes the affinity.

Unfortunately, as the frontier of each query will NOT be known until the query execution is finished, we cannot precompute the affinity for a batch of queries without evaluating them. Hence, we cannot solve the above optimization problem precisely in advance. To address this challenge, we need a more proactive approach. Next, we present a heuristic to approximate the best alignment.

Heuristic-based Solution. First, we observe that *the distribution of frontier sizes tends to be highly biased across iterations*, thanks to the power-law nature of many real-world graphs. Figure 7 reports the frontier sizes across iterations during the evaluation of a few vertex-specific queries on two real-world graphs.

From the results, we can easily find some patterns in evaluating vertex-specific queries on power-law graphs: in the early iterations, the frontier grows exponentially, which we call the *expansion phase*. After reaching the "peak", the frontier starts to shrink quickly and steadily until it becomes empty, referred to as *stabilization phase*. The several iterations around the "peak" often dominate overall size of frontiers for the whole query evaluation.

With the above observations, we decide to focus the alignment on these dominating iterations, referred to as "*heavy iterations*". The rationale behind this decision is two-fold:

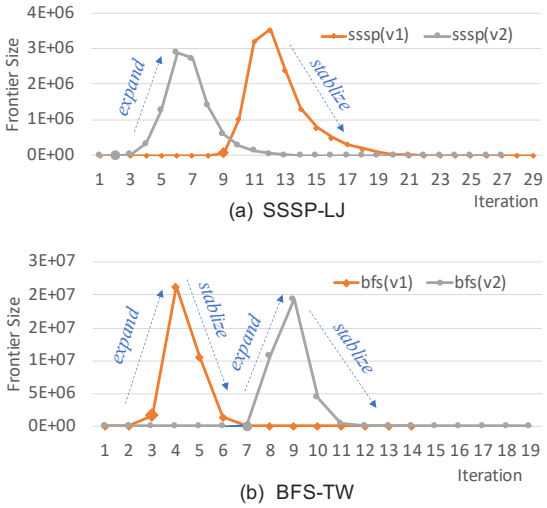

Figure 7: Frontier Size Distribution across Iterations.

- First, heavy iterations expose more opportunities for shared graph accesses. The larger the frontiers are, the more likely that they overlap. In the extreme case, when the (separate) frontiers include all vertices, they are perfectly overlapped.
- Second, as the vertex activations in the heavy iterations often dominate the total number of vertex activations for the whole evaluation, their alignments could make a significant impact on the overall alignment.

To demonstrate the effectiveness of heavy iteration alignment in improving the overall affinity, we manually delayed the "faster" queries in Figure 7 such that their "peaks" align with those in the "slower" queries. As a result, we observed that the affinity value gets improved from −0.11 to 0.34 and from 0 to 0.47, respectively.

One can quantify the heavy iterations based on different metrics, such as the ranking of frontier sizes across iterations. However, regardless the metric being used, just like affinity, heavy iterations are *unknown* before the query evaluation. In fact, one may choose to detect the heavy iterations dynamically during the query evaluation and use that information to guide the alignments at runtime, for example, pausing the evaluation of queries whose heavy iterations have arrived, then resuming them after all the "slowest" query has reached its heavy iterations. Though the idea sounds promising, it has a caveat—to "pause and resume" the iterative evaluation of some queries, the system needs to keep "their contexts"—their individual frontiers. This requires to a design similar to the two-level frontiers (see Figure 5-(b)). As discussed earlier, this design is inferior to the query-oblivious frontier in terms of performance.

To work around the above dilemma, we propose to proactively approximate the *"arrival time" of heavy iterations*—the iteration that marks the beginning of heavy iterations. The key insight behind our arrival time approximation is the correlation between frontier size and the activation of high-degree vertices:

> *When evaluating a vertex-specific query on a power-law graph, the frontier size often grows sharply once a high-degree vertex is activated.*

Table 4: Arrival Time of Heavy Iterations

Query	Arrival Time	Query	Arrival Time
sssp(v1)	iter. 9	bfs(v1)	iter. 3
sssp(v2)	iter. 2	bfs(v2)	iter. 7

To demonstrate the above phenomenon, Table 4 lists the first iteration where at least one of the top-4 high degree vertices is activated for the four queries used in Figure 7. These iterations are also highlighted with larger marks in Figure 7, which clearly indicate the beginning of (relatively) heavy iterations.

Based on the above discussion, we only need to identify the first iteration where a high-degree vertex is activated. In fact, given a high-degree vertex v_h (based on a threshold), it takes i iterations for it to be activated, where i is the least number of hops from the source vertex v in query $q(v)$ to the high-degree vertex v_h. Such information can be pre-computed simply by running a BFS query on the high-degree vertex v_h, that is, $\texttt{bfs}(v_h)$. Note that, for directed graphs, the BFS query should run on the edge-reversed graphs, since the goal is to get the least number of hops from every other vertex to the high-degree vertex, not the other way around.

Figure 8 summarizes our reasoning—reducing the inter-iteration alignment to the problem of BFS queries on high-degree vertices.

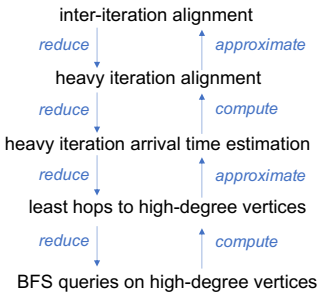

Figure 8: Flow of Solving Inter-Iteration Alignment.

Next, we present the algorithm of inter-iteration alignment (see Figure 9). First, some one-time preparation is needed: (i) identifying the top-K high-degree vertices in terms of the out-degree (due to the use of push model) at Line 2; (ii) reversing the edges' directions if the graph is directed (Line 3); (iii) running a BFS query on each selected high-degree vertex to find the least number of hops from an arbitrary vertex v_i to each high-degree vertex v_h (Line 4-5).

After the preparation, the algorithm is ready to compute the alignment vector I for a give query batch $\mathcal{B}$. First, it computes the least hop number to the closest high-degree vertex for each query $q(v_i)$, stored in $\texttt{closestHV}[v_i]$ (Line 9-10). Then, it finds the largest value among $\texttt{cloestHV}[v_i]$ (Line 11), which essentially is the latest time of reaching a high-degree vertex (i.e., arrival time) for a query in the batch, stored in `latestInBatch`. Finally, based on the difference of arrival time relative to the latest (`latestInBatch`), it calculates the alignment vector I (Line 12-13).

From another perspective, the alignment delays the start time of certain queries in the evaluation batch, thus we also refer to the above technique as *delayed start*. As shown later in the evaluation, comparing to the ground truth—the optimal alignment, the accuracy of the above heuristic-based alignment is quite high (usually off by at most 2 iterations) and also its performance is close to that with the optimal alignment. See more details in Section 4.

```
1  /* preparation for graph G */
2  HV = getTopHighOutDegreeVertices(G, K)
3  G_r = getEdgeReversedGraph(G)
4  for each vertex v_h in HV
5     leastHops[v_1 - v_max][v_h] = bfs (G_r, v_h)
6
7  /* find alignment vector I for query batch B */
8  getAlignment(B) {
9     for each query q(v_i) in B
10       closestHV[v_i] = min { leastHops[v_i][v_h] | ∀ v_h ∈ HV }
11    latestInBatch = max { closestHV[v_i] | ∀ q(v_i) ∈ B }
12    for each query q(v_i) in B
13       I [v_i] = latestInBatch − closestHV[v_i]
14 }
```

Figure 9: Pseudocode for Inter-Iteration Alignment.

Next, we will move our discussion of alignment to the most coarse-grained level—query batching.

3.4 Alignment-Aware Batching

In the prior section, we align (local) iterations of different queries to improve the sharing of graph accesses (measured by affinity). In fact, we may achieve similar or even better effects by grouping queries whose iterations are better aligned into the same batch. For example, it is better to put $\texttt{sssp}(v_1)$ (in Table 1) and $\texttt{sssp}(v_2)$ into the same batch, rather than $\texttt{sssp}(v_2)$ and $\texttt{sssp}(v_8)$ (in Table 2), as the former exposes a better alignment at the iteration level.

Based on the above intuition, we propose *affinity-oriented query batching*. The goal is to maximize the affinity for queries in a batch through the management of batching policy.

Affinity-Oriented Query Batching. By default, all queries in the evaluation buffer are processed in the order that they are received. Though intuitive, this first-come first-serve policy may produce query batches with low affinity. To avoid this, affinity-oriented batching creates batches based on the affinity among queries, which can be approximated as detailed in the prior section. However, a simple affinity-oriented batching, in theory, may postpone the processing of some queries—those exhibit poor affinity with most queries—with an unbounded delay. To avoid this caveat, we limit the number of queries considered each time for affinity-oriented batching using a threshold B_w. That is, every B_w queries in the buffer are scheduled together, referred to as a *batching window*.

Figure 10 illustrates the idea of affinity-oriented query batching. First, the earliest received B_w queries in the buffer are selected and ranked by their least number of hops to the closest high-degree vertex (i.e., `closestHV[]` in Figure 9). Then, every consecutive $|\mathcal{B}|$ ranked queries in the batch window are selected to form an evaluation batch. Note that array `closestHV[]` is pre-computed just like that used in inter-iteration alignment.

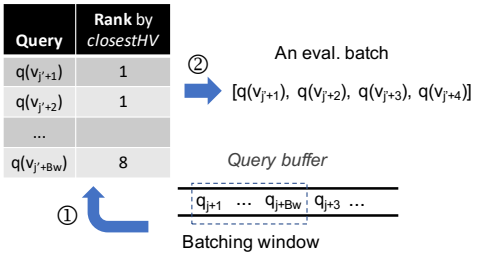

Figure 10: Affinity-Aware Query Batching.

Connection with Inter-Iteration Alignment. Note that unlike the intra-iteration alignment which is orthogonal to the later two alignment techniques, the relation between the two inter-iteration alignments and affinity-oriented batching are not fully orthogonal. This is due to the fact that both aim at improving the affinity defined at the iteration level. In other words, if affinity-oriented batching has been employed, then the extra benefits from inter-iteration alignment might be limited, even though in theory, each alignment technique may provide its unique benefits.

So far, we have introduced the alignments at all three levels. Next, we briefly explain their implementations.

3.5 Implementation

We implemented `Glign` on top of the popular in-memory graph processing engine `Ligra` [28]. In fact, thanks to its high-efficiency, `Ligra` also serves as the base for two recently proposed concurrent graph systems: `Congra` [20] and `Krill` [6]. One of our goals is to implement `Glign` as a transparent runtime system, thus keeping the original programming interface of `Ligra` mostly untouched. For example, developers can still call functions `EdgeMap()` and `VertexMap()` for traversing edges and vertices, and `VertexSubset` remains to be the frontier representation.

Under the hood, `Glign` maintains the query-oblivious frontier and leverages the original parallelization supports from `Ligra` to process each query in parallel. To support batching, we extended `Ligra` to let it consume a query buffer based on the batch size B and the batching policy (e.g., affinity-oriented batching). At the beginning of an iteration, `Glign` first checks the alignment vector I (in Section 3.3) to decide if some queries need to be started at the current iteration. For better locality, the memory layout of vertex values is implemented as a single array and the value of vertex v_j for query q_i can be accessed via `ValArray[`v_j`*B+i]`, where B is the batch size. When the graph is loaded into the memory for the first time, Glign will automatically compute the least hops from each vertex to the closest high degree vertex (`closestHV[]`), which will later be used to guide the alignments.

4 EVALUATION

This section evaluates the effectiveness of the proposed alignment techniques and the efficiency of `Glign`.

4.1 Methodology

First, we set up two baselines for comparing with `Glign`: (i) `Ligra-S` and (ii) `Ligra-C`. The former evaluates the queries in a batch one after another, using `Ligra` [28]—a state-of-the-art in-memory graph processing engine. Note that `Ligra` itself processes each single query in parallel. `Ligra-C` extends `Ligra` to support concurrent query evaluation using both unified and separated frontiers (see Section 3.2), representing a design that has been adopted by the existing concurrent graph processing systems [6, 13, 42].

Table 5: Methods in Evaluation

Method	Brief Description
`Ligra-S`	Eval. queries in batch one by one w/ `Ligra` [28]
`Ligra-C`	Eval. queries in batch simultaneously w/ `Ligra` [28]
`GraphM` [42]	A high-throughput concurrent graph system
`Krill` [6]	A compiler & runtime for concurrent graph processing
`Glign-Intra`	`Glign` with only intra-iteration alignment
`Glign-Inter`	`Glign-Intra` + inter-iteration alignment
`Glign-Batch`	`Glign-Intra` + affinity-oriented batching
`Glign`	`Glign` with all proposed alignment techniques

Also, we compare `Glign` in general with two state-of-the-art concurrent graph processing systems that are publicly accessible: `GraphM` [42] and `Krill` [6]. To show the contributions of different techniques, `Glign` is configured differently as listed in Table 5. In addition to the above systems, we also tested a system design that exploits query-level parallelism—each concurrent query is evaluated using the serial implementation from BGL [29], while different queries are processed on different threads. However, we found it ran slower than our baseline `Ligra-S` in most cases tested.

Queries. We evaluated five types of graph queries, including BFS (*breadth-first search*), SSSP (*single source shortest path*), SSWP (*single source widest path*), SSNP (*single source narrowest path*), and Viterbi. Table 6 lists their vertex functions in pseudo-code.

All queries are vertex-specific in that they start from one source vertex and compute property values for all vertices in the graph. To generate the query set for each type of query, we followed a sampling strategy similar to the one used by Qi and others [21]. First, the graph vertices are divided into disjoint bins based on their distances (hops) to the (top-4) high-degree vertices. Then, these bins are scanned in rounds, and in each round a vertex is randomly picked from each bin, until 512 vertices are selected, which serve as the source vertices of our queries. In this way, the selected queries provide a better coverage of the entire graph structure. We assume all the 512 queries are already in the buffer when the systems start to process them. This allows us to focus on evaluating the throughput of the concurrent systems. One can also add the query arrival time information, with which the latency of processing each query could also be inferred. We leave such latency study for future work.

Besides grouping queries of the same types into the same query buffer (i.e., homogeneous query buffer), we also generated a query buffer of mixed types of queries, randomly selected with types of BFS, SSSP, SSWP, and SSNP. We refer to this scenario as "Heter".

By default, we set the query batch size to 64. For evaluating the impacts of batch size, we changed the batch size from 2 to 128.

Graph Data Sets. We primarily evaluate `Glign` on power-law graphs, which include five real-world graphs. For completeness, we also evaluate `Glign` on a couple of road networks, which are more like planar graphs. Their basic properties are summarized in

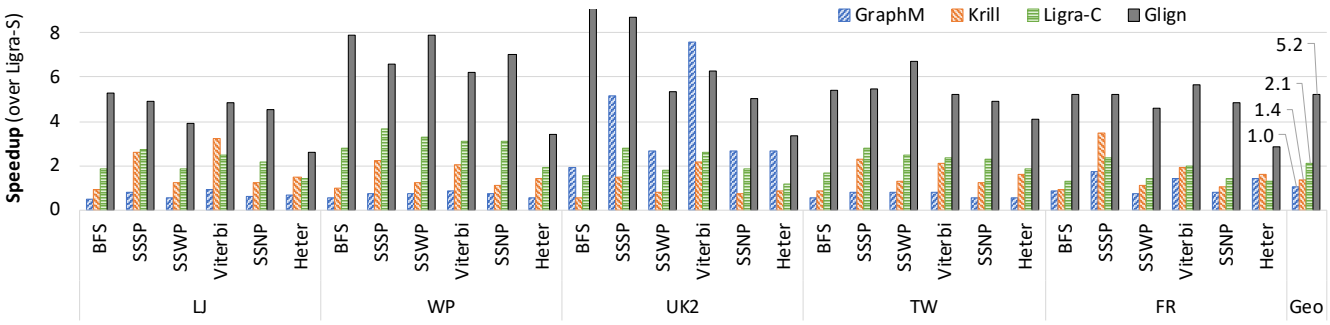

Figure 11: Overall Performance.

Table 6: Vertex Functions of Graph Queries

Bench.	Pseudo-code of Vertex function $f(s)$
BFS	**for** each out-neighbor d of s level(d) = min { level(d), level(s) +1 }; **if** level(d) changed **then** add d to $frontier$;
SSSP	**for** each out-neighbor d of s dist(d) = min { dist(d), dist(s) + w(s, d) }; **if** dist(d) changed **then** add d to $frontier$;
SSWP	**for** each out-neighbor d of s wide(d) = max { wide(d), min { wide(s), w(s, d) } }; **if** wide(d) changed **then** add d to $frontier$;
Viterbi	**for** each out-neighbor d of s viterbi(d) = max { viterbi(d), viterbi(s) / w(s, d) }; **if** viterbi(d) changed **then** add d to $frontier$;
SSNP	**for** each out-neighbor d of s narrow(d) = min { narrow(d), max { narrow(s), w(s, d) } }; **if** narrow(d) changed **then** add d to $frontier$;

Table 7. The number of edges ranges from 69M to 3.6B, and the diameter ranges from 10 to 9100.

Table 7: Graph Statistics

Graph	Abbr.	Directed	|V|	|E|	Avg. deg.	Dia.
LiveJournal [3]	LJ	Yes	4.8M	69M	14.2	13
Wikipedia [2]	WP	Yes	14M	437M	32.2	10
UK-2002 [5]	UK2	No	19M	524M	28.3	45
Twitter [12]	TW	Yes	42M	1.5B	35.3	15
Friendster [1]	FR	No	125M	3.6B	28.9	38
roadNet-CA [24]	RD-CA	No	2.0M	5.5M	2.81	849
roadNet-USA [24]	RD-US	No	24M	58M	2.41	9100

The experiments are conducted on a 32-core Linux server that equips with Intel Xeon E5-2683 v4 CPU (LLC size: 40MB) and 512GB memory. The application is compiled with g++ 6.3 and runs on CentOS 7.9.

Next, we will first report the overall performance, followed by more detailed evaluation of each alignment technique.

4.2 Overall Performance

Table 8 shows the total execution time of evaluating a buffer of 512 queries using `Ligra-S`, while Figure 11 reports the speedups

Table 8: Time of Evaluating 512 Queries using `Ligra-S`

	LJ	WP	UK2	TW	FR
BFS	66s	300s	334s	1104s	3646s
SSSP	176s	579s	986s	2332s	10076s
SSWP	97s	392s	507s	1734s	4580s
Viterbi	224s	499s	1388s	2129s	7926s
SSNP	99s	372s	493s	1630s	4440s
Heter	107s	398s	567s	1819s	5858s

of the other systems over `Ligra-S`. From the results, we find that `Glign` clearly outperforms the other systems in almost all the cases (except for the case UK2-Viterbi). The highest speedup it reaches is 9.4× (in the case of UK2-BFS). On average, `Glign` achieves 5.2× speedup over the baseline `Ligra-S`.

Among the other systems, `GraphM` exhibits similar performance as `Ligra-S`. Note that, unlike the other systems in our evaluation, `GraphM` is not built on top of `Ligra`, instead, it is built on top of `GridGraph` [43], a system mainly designed for out-of-core graph processing. This difference in the base system selection could be one of the reasons that cause `GraphM` to perform worse than the other concurrent graph systems used in our evaluation.

`Krill` and `Ligra-C` both achieve a substantial average speedup over `Ligra-S` (1.4× and 2.1×), confirming the general benefits of concurrent query evaluation. However, both perform worse than `Glign`, though all the three systems share the same base system (`Ligra`). We believe this is mainly due to the locality improvement brought by the alignment techniques that `Glign` employed.

To confirm the above speculation, we measured the last-level cache (LLC) misses using the `perf` tool. The results are listed in Table 9. To save space, we report the results mainly for two graphs. From the results, we find that `Glign` incurs significantly fewer LLC misses than the other methods. For example, on LJ graph, `Glign`'s LLC misses is only 12%, 21%, 5%, and 23% of those (on average) incurred by `Ligra-S`, `Ligra-C`, `GraphM`, and `Krill`, respectively. These significant LLC miss reduction echos the substantial speedups brought by `Glign` as shown earlier in Figure 11.

In the following, we will break down the performance gains of `Glign` by evaluating each alignment technique.

Table 9: LLC Misses (in billions)

		Ligra-S	Ligra-C	GraphM	Krill	Glign
LJ	BFS	9	11	44	15	**1**
	SSSP	28	15	77	15	**4**
	SSWP	21	12	52	15	**2**
	Viterbi	53	20	91	8	**4**
	SSNP	21	11	52	15	**2**
	Heter	23	15	45	11	**4**
Mean		26	14	60	13	**3**
TW	BFS	302	245	1083	394	**26**
	SSSP	621	237	1698	399	**53**
	SSWP	545	217	963	397	**29**
	Viterbi	757	242	1622	171	**48**
	SSNP	540	214	1202	396	**36**
	Heter	563	252	1201	241	**52**
Mean		555	235	1295	333	**41**

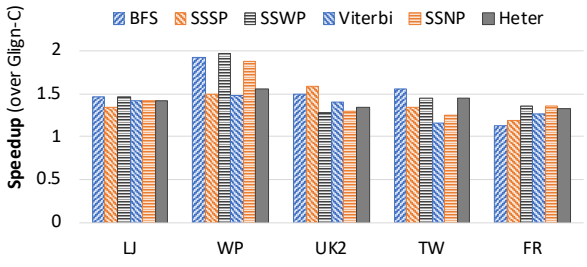

Figure 12: Speedups of `Glign-Intra` over `Ligra-C`

4.3 Intra-Iteration Alignment

In this section, we evaluate the proposed query-oblivious frontier (Section 3.2) and compare it with the two-level frontier design (i.e., unified and separate frontiers) that is employed by some of the existing concurrent query processing systems. Both designs ensure synchronized frontier traversal—the key to intra-iteration alignment. In our setting, `Ligra-C` employs the two-level frontier design, while `Glign-Intra` uses the query-oblivious frontier (other alignment techniques are disabled).

First, we have verified the correctness of all the query results produced by `Glign-Intra`, thus experimentally demonstrated the correctness of this new frontier design with real-world large data, complimenting the theoretical proof in Section 3.2.

Second, in terms of performance, Figure 12 reports speedups of `Glign-Intra` over `Ligra-C`. The results show that `Glign-Intra` yields consistent speedups across different queries and graphs, which range from 1.13× to 1.96×.

In addition, we also collected the LLC misses of `Glign-Intra` which may offer some more direct evidences on the effectiveness of query-oblivious frontier. The results are reported in Table 10. From the results, we find that `Glign-Intra` can consistently reduce the LLC misses under all the evaluated cases, and the average reduction is quite substantial—its LLC misses are only around 30% on average of those incurred by the baseline `Ligra-C`. The reduction mainly comes from the elimination of separate frontiers and the two-level frontier checking used by the baseline and other existing concurrent graph systems. These results, to a large extent, explain `Glign-Intra`'s speedups shown in Figure 12.

Table 10: LLC Misses Reduction by `Glign-Intra`

(Numbers are the ratios between the LLC misses of `Glign-Intra` and the LLC misses of `Ligra-C`)

	LJ	WP	UK2	TW	FR
BFS	22%	24%	13%	23%	32%
SSSP	33%	37%	12%	34%	36%
SSWP	32%	27%	17%	28%	24%
Viterbi	34%	39%	19%	36%	31%
SSNP	32%	28%	17%	31%	21%
Heter	35%	42%	22%	31%	31%
Geomean	31%	32%	16%	30%	29%

Table 11: Memory Footprint Breakdown (64 queries)

	LJ		TW	
	Ligra-C	Glign-Intra	Ligra-C	Glign-Intra
Graph	545MB	545MB	11,400MB	11,400MB
Vertex Value	1,180MB	1,180MB	10,200MB	10,200MB
Frontier	296MB	4.6MB	2,540MB	39.7MB

Finally, to get a sense of the memory reduction brought by the query-oblivious frontier in comparison to a two-level frontier, we profiled the memory footprints. Table 11 reports the sizes of major data structures in `Ligra-C` and `Glign`: (i) the graph topology data, the values of all vertices, and the frontier (as a labeling array). Note that even though the frontier is only a relatively small portion of the total memory footprint, it is accessed entirely in every iteration. In comparison, only some parts of the graph and some vertex values are accessed in each iteration. The results show that the frontier size is reduced dramatically with query-oblivious frontier, which leads to the LLC misses reduction as shown in Table 10.

4.4 Inter-Iteration Alignment

To show the benefits of inter-iteration alignment, we compare `Glign-Inter` with `Glign-Intra`.

Performance. Figure 13 reports the speedups of `Glign-Inter` over `Glign-Intra`. Overall, `Glign-Inter` achieves better performance in all evaluated cases, except for WP-SSWP and WP-SSNP. The speedups range from 0.89× to 2.95×. This demonstrates the benefits of our proposed inter-iteration alignment technique—*delayed start*. In general, we found `Glign-Inter` does not perform significantly better on WP graph. The reason might be related to the fact that WP has a relatively smaller diameter (see Table 7). Smaller diameters imply that the vertices selected as the source vertices of the queries tend to be closer to each other in terms of hops. Based on the discussion in Section 3.3, queries with closer source vertices tend to align better (i.e., yielding better affinity) during their evaluation. As a result, there is less room for inter-iteration alignment to improve.

Affinity. To get a deeper understanding of the improvements, we also collected the affinity values (see Definition 3.4). The results are shown in Figure 14. Note that we set the Y-axis to 1 − *affinity* because, for a batch of 64 queries, the affinity value tends to be

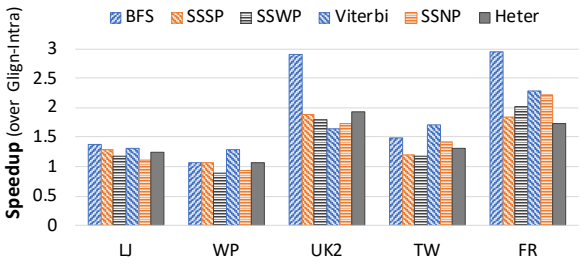

Figure 13: Speedups of `Glign-Inter` over `Glign-Intra`

close to 1, using 1 − *affinity* can better reflect its significants. In fact, 1 − *affinity* reflects *how much different frontiers are misaligned*, thus the lower the value is, the better alignment we get. The results show that `Glign-Inter` substantially reduces the divergence, from 0.068 to 0.041 on average. The highest reductions happen to BFS on UK2 and FR graphs, which match well with the top two best speedups (2.91× and 2.95×) reported in Figure 13. The results also show that divergence for `Glign-Intra` is already very low in the case of WP graph, leaving little room for further improvements. This explains the limited speedups achieved by `Glign-Inter` on this graph (see Figure 13).

In addition, we report the LLC miss reduction by `Glign-Inter` over `Glign-Intra` in Table 12. In general, the results echo well the above findings. For example, the highest cache miss reductions also happen to BFS on UK2 and FR graphs (37% and 32%) and the reduction ratio on WP graph is the least.

Table 12: LLC Misses Reduction by `Glign-Inter`

(Numbers are the ratios between the LLC misses of `Glign-Inter` and the LLC misses of `Glign-Intra`)

	LJ	WP	UK2	TW	FR
BFS	67%	96%	37%	68%	32%
SSSP	73%	91%	53%	77%	51%
SSWP	73%	101%	39%	76%	48%
Viterbi	64%	69%	38%	60%	37%
SSNP	74%	97%	41%	67%	42%
Heter	64%	81%	31%	65%	51%
Geomean	69%	88%	39%	69%	43%

Heuristic v.s. Ground Truth. To examine the effectiveness of the proposed heuristic, we profiled the ground-truth best alignments for 512 sampled batches, where each batch consists of only two queries. The best alignment of each batch is found by exhaustively trying all possible alignments and calculating the corresponding affinity value. Table 13 summarizes our findings. Among the 512 batches, our heuristic finds the best alignments in 33.6% of them, and the difference with the optimal alignment is within 2 iterations for over 95% cases. As to the speedups, the best alignments outperform ours (1.50× vs 1.45×), indicating that extra room exists for further improving the performance via alignments.

Profiling Costs. As discussed in Section 3.3, our heuristic requires profiling—running BFS queries on (four) high-degree vertices. Note that the profiling happens at the beginning when the system and graph is set up. It is a one-time effort for each graph, with benefits applying to different types of queries that run on the graph. Table 14 lists the profiling costs on two graphs (LJ and TW), compared to the query evaluation time on the same graphs. During the concurrent query evaluation, accessing the profiling result (a table lookup) is quick and the cost is negligible.

Table 13: Ground Truth Study of `Glign-Inter`

			Speedup		
Diff	Cnt	Ratio	Glign-Intra	Glign-Inter	Best-Align
0	172	33.6%	1.36×	1.51×	1.51×
1	217	42.4%	1.32×	1.42×	1.51×
2	102	19.9%	1.20×	1.27×	1.46×
3	20	3.9%	1.13×	1.20×	1.47×
4	0	0.0%	N/A	N/A	N/A
5	1	0.2%	1.62×	1.52×	1.56×
Sum./Avg.	512	100%	1.30×	1.41×	1.50×

Table 14: Profiling Costs

		LJ	TW
Profiling Cost		0.20s	3.84s
Query Eval. Cost (batch size:64, `Glign`)	SSSP	4.47s	53.40s
	BFS	1.56s	25.51s

4.5 Alignment-Oriented Batching

To demonstrate the benefits of alignment-oriented batching, we compare `Glign-Batch` with `Glign-Intra`. Figure 15 reports the speedups of `Glign-Batch` over `Glign-Intra`. These speedups are slightly higher than those achieved by `Glign-Inter` (Figure 13). This is expected as both alignments essentially explore the same affinity opportunities. The additional improvements indicate that the alignment opportunities across queries (in the query buffer) are slightly higher than those within a single evaluation batch, which is also confirmed by the affinity differences between `Glign-Inter` and `Glign-Batch` as shown in Figure 14.

4.6 Impacts of Batch Size

To understand the sensitivity of `Glign` to a basic parameter—the batch size. We changed the batch size from 2 to 128. Note that 128 is the largest value `Glign` can achieve based on the memory capacity of our machine and the size of the evaluated graphs.

Figure 16 reports the results using four types of queries and two input graphs. Most curves in the figure follow a similar trend, that is, the speedup first grows as the batch size increases, until the batch size reaches around 64, then the speedup starts to drop. The upward trend indicates that increasing the degree of concurrency tends to be beneficial, while the downward trend indicates there is a limit for the benefit—the memory pressure also increases as the batch size increases, which eventually would curb the gain.

4.7 Performance on Road Networks

Though `Glign` is primarily designed for processing power-law graphs, for completeness, we also report its performance on some

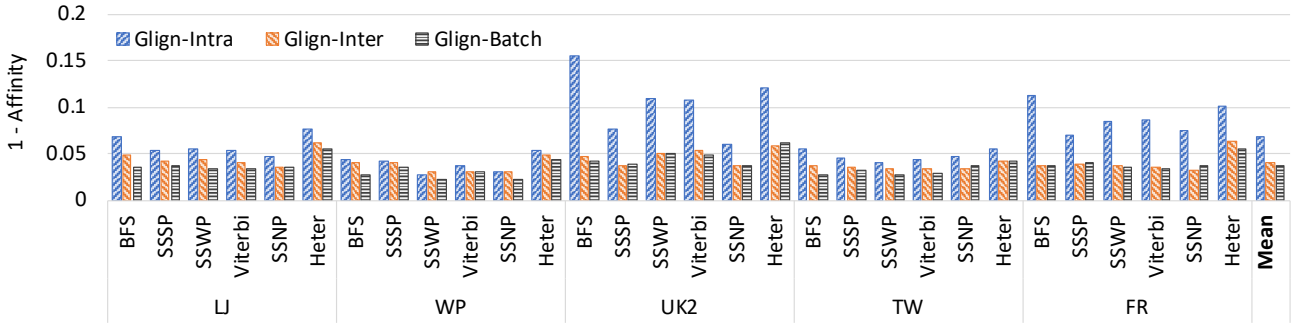

Figure 14: Affinity Comparison: `Glign-Intra` **vs** `Glign-Inter` **vs** `Glign-Batch`

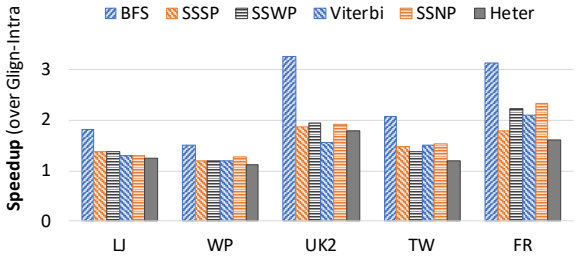

Figure 15: Speedups of `Glign-Batch` **over** `Glign-Intra`

Table 15: Performance on Road Networks

	RD-CA			RD-US		
	SSSP	BFS	SSWP	SSSP	BFS	SSWP
Ligra-S	369.8s	239.5s	219.6s	15224s	7916s	1298s
Ligra-C	2.91×	5.30×	1.45×	1.25×	2.66×	0.30×
Glign-Intra	6.08×	9.31×	2.99×	2.04×	14.67×	1.86×
Glign-Inter	6.85×	10.05×	3.15×	1.75×	13.51×	1.25×
Glign-Batch	8.36×	11.15×	3.66×	2.52×	15.37×	1.91×
Glign	8.90×	12.41×	3.64×	2.77×	16.91×	1.29×

road networks (RD-CA and RD-US). Table 15 reports the speedups of `Glign` and their variants over our baseline `Ligra-S`. First, the results show that `Glign-Intra` still achieves good or even higher speedups (e.g., 9.3× and 14.7× speedups for BFS). This is due the fact that, typically, only a small portion of the road network needs to be accessed in each iteration, which makes the costs to access frontier(s) relatively higher. On the other hand, the extra benefits brought by the inter-iteration alignment and alignment-oriented batching are more limited. This is because evaluation on such graphs often fails to yield sufficiently "heavy" iterations, making the affinity issue less of a concern.

4.8 Comparison with iBFS

Finally, we compare `Glign` with `iBFS` [13]—a specialized graph system dedicated to concurrent BFS queries. It is an early work that groups BFS query instances and leverages shared frontier traversal, which resembles affinity-oriented batching. However, there are a few key differences between the two. First, `iBFS` maintains both the unified and separate frontiers to achieve synchronized frontier traversal, just like `Ligra-C` and `Krill`. In comparison, `Glign` uses unified frontier only (i.e., query-oblivious frontier); Second, to group BFS queries, `iBFS` uses a different heuristic based on the out-degrees of source vertices. In particular, it requires two conditions for query grouping: (i) out-degrees of source vertices should be less than p; and (ii) the source vertices must connect to at least one common vertex whose out degree is greater than q. In comparison, `Glign` groups queries only based on the affinity (or number of hops to a high-degree vertex). Last but not least, `iBFS` does not support inter-iteration alignment. To experimentally compare the two, we have implemented the heuristic of `iBFS` in `Ligra-C`. Note that the original iBFS work is implemented for the GPU platform.

Table 16: Comparison with `iBFS`

	iBFS	Glign-Intra	Glign-Batch
LJ	16.6s	0.98×	1.78×
WP	41.1s	0.92×	1.45×
UK2	130.8s	0.95×	3.17×
TW	276.3s	1.08×	2.10×
FR	2465.1s	1.02×	3.04×

Table 16 reports the performance of using `iBFS` for evaluating 512 BFS queries, and the speedups of `Glign-Intra` and `Glign-Batch` over `iBFS`. Overall, we find that the performance of `iBFS` is similar to `Glign-Intra`, but substantially slower than `Glign-Batch`. On average the gap between `iBFS` and `Glign-Batch` is between 1.45× and 3.17×. A further examination reveals that the heuristic of `iBFS` is too strict—it works better when there are an extremely large number of queries involved (e.g., querying all vertices in the graph).

5 RELATED WORK

This section provides a more detailed summary of the works on multi-query graph processing as well as some other relevant works.

Targeting distributed platforms, `Seraph` [33, 34] is an early work that provides system-level supports for concurrent graph query processing. Its key idea is to decouple the graph structure from query-specific data to allow concurrent query evaluations to *share the common graph* structure data. Some other distributed systems for concurrent graph query processing include `MultiLyra` [16] and `BEAD` [17], both of which support efficient batched query evaluation

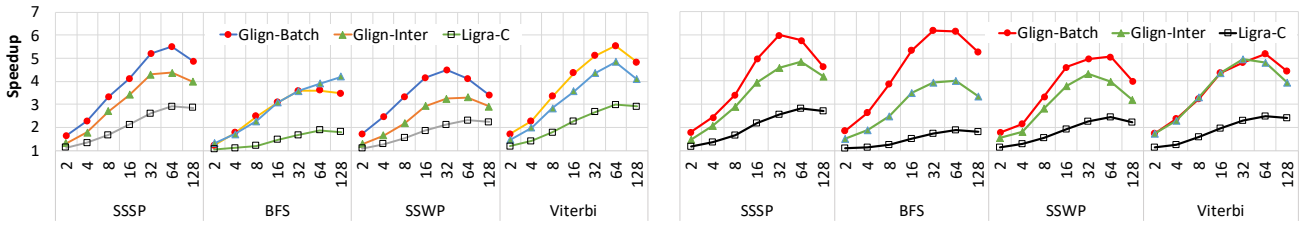

Figure 16: Impacts of Query Batch Size (Left: LJ graph, Right: TW graph)

with graph sharing and frontier sharing in order to amortize the communication costs among computation nodes in the cluster.

For single-machine graph processing with out-of-core processing supports, some prior works include CGraph [38] and GraphM [42], both of which follow the idea of graph sharing as in Seraph. The key insight of both works is to exploit the temporal and spatial locality of shared graph accesses. To achieve that, they first partition the graph (as well as the vertex value array) so that each partition is small enough to fit into the cache, then they load one partition each time and process all jobs (queries) relevant to this partition. Comparing to their "partition-centric" approach, our approach is more "iteration-centric", though both of them share the same goal—improving the data locality.

For in-memory graph processing, one early work is Congra [20], which is built on top of Ligra [28]. Its main goal is to maximize the memory bandwidth by forking processes for new queries through the guidance of a scheduler. It needs offline profiling on each graph query to get the memory bandwidth and scalability characteristics. Since each query is evaluated by a separate process, Congra does not exploit graph sharing or frontier sharing. Unlike the others, SimGQ [32] exploits the shared subcomputations of queries in a batch of concurrent queries and reuses some computation results proposed in VRGQ [10] to improve the efficiency. These techniques exploit shared computations, thus are orthogonal to our techniques which exploit shared graph accesses. Finally, Krill [6] is a more recent work that not only exploits graph sharing, but also gains benefits from an efficient management of *property data* that are computed during the evaluation. In comparison, our work supports both *graph sharing* and *frontier sharing* among concurrent queries while also achieves improvements in alignment of graph traversals and thus significant reductions in LLC misses over Krill. Like the above systems, our system Glign also targets in-memory graph processing, however, Glign explores three levels of alignments which are not seen in any of these existing systems.

Following up the Ligra [28] framework, two more recent graph programming systems, GraphIt [37, 40] and Julienne [7], have been proposed for optimizing single query processing. In particular, GraphIt designs a DSL that provides custom scheduling functions for exploring various optimization opportunities. In comparison, Julienne is specialized for bucketing-based algorithms like k-core and approximate set-cover which cannot be supported efficiently in Ligra. Their techniques are orthogonal to those in our work which focus on the performance of concurrent queries. However, to incorporate our alignment techniques, these systems need to be extended to support concurrent query evaluation, which appears to be more challenging than extending the Ligra framework, due to the involvement of a DSL and the consideration of bucketing-based algorithms, respectively. Some other recent works offer supports for hypergraphs [27] and streaming graphs [8, 31]—scenarios that our work does not cover currently. It would be an interesting topic to explore concurrent query evaluation under such schemes.

There are recent works on graph accelerators (GraphPulse [22], JetStream [23], and LCCG [41]). GraphPulse is an asynchronous graph processing accelerator for static graphs. JetStream supports streaming graphs and incremental computations. It also exploits the monotonicity property of iterative graph queries, but for a different purpose than our work. The property is leveraged to guarantee the correctness of incremental computations. Neither GraphPulse or JetStream addresses the concurrent query evaluations scenario. LCCG is a graph accelerator that supports concurrent graph jobs by utilizing a topology-aware approach with new hardware units. It remains an interesting open question how our proposed alignment techniques can be integrated into the graph accelerators. One of the first steps should be extending the above accelerators so that they can handle multiple queries simultaneously.

Finally, there are many works aimed at improving the memory locality for a single query evaluation [19, 36, 39]. To carryover these improvements to concurrent queries, they must be combined with an approach like Glign.

6 CONCLUSION

This work reveals a major performance issue in concurrent graph processing—alignment of graph traversals. It addresses this issue at three levels. First, it proposes the query-oblivious frontier to achieve synchronized frontier traversal within each global iteration. Second, it introduces a heuristic-based solution based a series of insights and observations to intelligently align the iterations of different queries and to group queries with different affinities. It integrates the proposed techniques into a runtime system called Glign. A full evaluation of Glign has confirmed the effectiveness of the proposed alignments and demonstrated superior performance over state-of-the-art concurrent graph processing systems.

ACKNOWLEDGMENTS

We thank all the reviewers for their very valuable feedback. This material is based upon the work supported in part by National Science Foundation Grants CCF-2028714, CCF-2002554 and CCF-1813173. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code of `Glign`, including the five concurrent query evaluation designs discussed in the paper and some graph benchmarks used in the experiments. In addition, this artifact provides bash scripts to compile `Glign` and reproduce the key experimental results reported in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Five designs of concurrent query evaluation schemes: `Glign`, `Glign-Intra`, `Glign-Inter`, `Glign-Batch`, and `Ligra-C`.
- **Program:** The concurrent query evaluation system `Glign` built on top of `Ligra`.
- **Compilation:** GCC 6.3.0
- **Data set:** There are seven graphs tested in the paper. We include the smaller ones (LiveJournal and roadNet-CA) in the artifact.
- **Run-time environment:** The system is developed and tested in Linux environment (CentOS 7.9).
- **Hardware:** The experiments in the paper were run on a machine with Intel Xeon E5-2683 v4 CPU and 512GB memory. 150GB disk space is enough for storing all graphs.
- **Experiments:** Bash scripts are included in `Glign-AE/apps` and `Glign-AE/results/scripts`. Detailed instructions are provided in `Glign-AE/README.md`
- **How much time is needed to prepare workflow (approximately)?:** 2-3 hours to prepare all graphs.
- **How much time is needed to complete experiments (approximately)?:** It takes around 2-4 hours to generate all data for the LiveJournal (LJ) graph. Note that collecting data for all the graphs reported in the paper may be very time-consuming. It is suggested that the reviewer first test the LJ graph and check the results. If time allows, the reviewer can test other larger graphs.
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to access. A file named `ASPLOS23_AE.zip`, containing the source files, scripts, and input query files, is available as a public repository on Zenodo (find its URL in [35].)

A.3.2 Hardware dependencies. To reproduce the results reported in the paper, we recommend running the artifact on Intel Xeon CPU with at least 32 cores. For smaller graphs like LJ, 64GB memory is enough, for larger graphs (FR), 400GB memory is required.

A.3.3 Software dependencies. We recommend that the artifact runs on CentOS 7, but other similar Linux distributions should also work. To compile and run the source code with scripts, users need GCC with Cilk support (GCC 7.4.0 or 6.3.0 or lower versions).

A.4 Installation

The compilation script is provided in `Glign-AE/apps`. Detailed instructions are provided in `Glign-AE/README.md`

A.5 Evaluation and expected results

The performance tests and profiling experiments could be run with the provided scripts (in `Glign-AE/apps`). See detailed instructions in `Glign-AE/README.md`. Scripts for collecting data reported in the paper are provided in `Glign-AE/results/scripts`). Note that the running times, speedups, and LLC misses may vary depending on the computing environments, but the trends should be similar.

REFERENCES

[1] 2013. Friendster network dataset. http://konect.cc/networks/friendster/. Accessed: 2022-01-02.

[2] 2013. Wikipedia links, english network dataset. http://konect.cc/networks/wikipedia_link_en/. Accessed: 2022-01-02.

[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 44–54.

[4] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30, 8 (1995), 207–216.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[6] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. 2021. Krill: a compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[7] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 293–304.

[8] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 918–934.

[9] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

[10] Xiaolin Jiang, Chengshuo Xu, and Rajiv Gupta. 2021. VRGQ: Evaluating a Stream of Iterative Graph Queries via Value Reuse. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 11–20.

[11] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 17–32.

[12] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[13] 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*. 403–416.

[14] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078* (2012).

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

[16] Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2019. Multilyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 349–358.

[17] Abbas Mazloumi, Chengshuo Xu, Zhijia Zhao, and Rajiv Gupta. 2020. BEAD: Batched Evaluation of Iterative Graph Queries with Evolving Analytics Demands. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 461–468.

[18] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[19] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 1–14. https://doi.org/10.1109/MICRO.2018.00010

[20] Peitian Pan and Chao Li. 2017. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 217–224.

[21] Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. 2013. Toward a distance oracle for billion-node graphs. *Proceedings of the VLDB Endowment* 7, 1 (2013), 61–72.

[22] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 908–921.

[23] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. 2021. JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1091–1105.

[24] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[25] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[26] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2016. *Large-scale graph processing using Apache Giraph*. Springer.

[27] Julian Shun. 2020. Practical parallel hypergraph algorithms. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 232–249.

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

[29] Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine, et al. 2002. *The boost graph library*. Vol. 243. Pearson India.

[30] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[31] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[32] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2020. SimGQ: Simultaneously evaluating iterative graph queries. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 1–10.

[33] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. 2016. Processing concurrent graph analytics with decoupled computation model. *IEEE Trans. Comput.* 66, 5 (2016), 876–890.

[34] 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*. 227–238.

[35] Xizhe Yin. 2022. *Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing*. https://doi.org/10.5281/zenodo.7173860

[36] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast Iterative Graph Computation: A Path Centric Approach *(SC '14)*. IEEE Press, 401–412. https://doi.org/10.1109/SC.2014.38

[37] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2019. Optimizing ordered graph algorithms with graphit. *arXiv preprint arXiv:1911.07260* (2019).

[38] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 441–452.

[39] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An Efficient Path-Based Iterative Directed Graph Processing System on Multiple GPUs *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 601–614. https://doi.org/10.1145/3297858.3304029

[40] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[41] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. 2021. LCCG: a locality-centric hardware accelerator for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[42] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[43] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. {GridGraph}:{Large-Scale} Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.

Received 2022-03-31; accepted 2022-06-16