Journal of Parallel and Distributed Computing 164 (2022) 12–27

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

SimGQ+: Simultaneously evaluating iterative point-to-all and point-to-point graph queries

Chengshuo Xu [*], Abbas Mazloumi, Xiaolin Jiang, Rajiv Gupta

University of California Riverside, Computer Science Department, United States of America

ARTICLE INFO

Article history:
Received 14 June 2021
Received in revised form 30 September 2021
Accepted 19 January 2022
Available online 1 February 2022

Keywords:
Batch of queries
Sharing computation
Amortizing overhead
Power-law graphs
Graph analytics

ABSTRACT

Graph processing frameworks are typically designed to optimize the evaluation of a single graph query. However, in practice, we often need to respond to multiple graph queries, either from different users or from a single user performing a complex analytics task. Therefore in this paper we develop SimGQ+, a system that optimizes simultaneous evaluation of a group of *vertex queries* that originate at different source vertices (e.g., multiple shortest path queries originating at different source vertices) and delivers substantial speedups over a conventional framework that evaluates and responds to queries one by one. Our work considers both point-to-all and point-to-point queries. The performance benefits are achieved via *batching* and *sharing*. *Batching* fully utilizes system resources to evaluate a batch of queries and amortizes runtime overheads incurred due to fetching vertices and edge lists, synchronizing threads, and maintaining computation frontiers. *Sharing* dynamically identifies *shared queries* that substantially represent subcomputations in the evaluation of different queries in a batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all queries in the batch. With four input power-law graphs and four graph algorithms SimGQ+ achieves speedups of up to 45.67× with batch sizes of up to 512 queries over the baseline implementation that evaluates the queries one by one using the state of the art Ligra system. Moreover, both batching and sharing contribute substantially to the speedups.

1. Introduction

Graph analytics is employed in many domains (e.g., social networks, web graphs, internet topology, brain networks etc.) to uncover insights by analyzing high volumes of connected data. An iterative algorithm updates vertex property values of active vertices in each iteration driving them towards their final stable solution. When the solution values of all vertices become stable, the algorithm terminates. It has been seen that real world graphs are often large with millions of vertices and billions of edges. Moreover, iterative graph analytics requires repeated passes over the graph till the algorithm converges to a stable solution. As a result, in practice, iterative graph analytics workloads are data-intensive and often compute-intensive. Therefore, there has been a great deal of interest in developing scalable graph analytics systems like Pregel [14], GraphLab [13], GraphIt [32], PowerGraph [5], Galois [18], GraphChi [10], Ligra [22], ASPIRE [26].

While the performance of graph analytics has improved greatly due to advances introduced by the above systems, much of this research has focused on developing highly parallel algorithms for solving a single iterative graph analytic query (e.g., SSSP(s) query computes shortest paths from a single source s to all other vertices in the graph) on different computing platforms including shared-memory systems, distributed clusters, and systems with accelerators like GPUs. However, in practice the following two scenarios involve multiple queries: (a) Single-User scenario as in Quegel [30], where the authors developed an analyzer for online shopping platform that frequently computes shortest-paths between some important shoppers in a large network extracted from online shopping data; and (b) Multi-User scenarios as in Congra [17] and [24] where the same data set is queried by many users. In both scenarios, machine resources can be fully utilized delivering higher throughput by simultaneously evaluating multiple queries on a modern server with many cores and substantial memory resources.

In this paper we develop a graph analytics system named SimGQ+, an extension of SimGQ [28], aimed at evaluating a batch of *point-to-all* and *point-to-point* queries received from users for different source vertices of a large graph. A point-to-all query has a single source vertex and has all other vertices as its destinations while a point-to-point query has a single destination. For example, for SSSP algorithm, we may be faced with the following batch of

[*] Corresponding author.
E-mail address: cxu009@ucr.edu (C. Xu).

https://doi.org/10.1016/j.jpdc.2022.01.007
0743-7315/© 2022 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

point-to-all queries: $SSSP(s_1)$, $SSSP(s_2)$, $\cdots\cdots$ $SSSP(s_n)$. Many other important algorithms belong to this category [11,7,6] etc. Our overall approach is as follows. Given an input graph and batch of vertex queries, we synergistically perform simultaneous evaluation of all queries in a batch to deliver results of all queries in a greatly reduced time. Essentially the synergy in evaluation of queries, that exists due to the substantial overlap between computations and graph traversals for different queries, is exploited to amortize the runtime overhead and computation costs across the simultaneously evaluated queries. Two techniques, *batching* and *online sharing*, are employed to simultaneously and efficiently evaluate a set of queries.

(a) Batching for Resource Utilization and Amortizing Overheads. Batching takes a group of queries, forming the batch, and simultaneously processes these queries to achieve higher throughput by fully utilizing system resources and amortizing runtime overhead (e.g., synchronization) costs across queries. Some prior works [30,24] process multiple queries simultaneously. In [24] authors process multiple queries but the solution is optimized specifically for BFS queries. Quegel [30] pipelines execution of a few queries delivering limited performance enhancement as shown in [15]. MultiLyra [15] performs batching on distributed systems and thus mainly derives benefits from amortizing cost of communication between machines. In contrast, SimGQ+ is capable of evaluating large batches (up to 512 queries) of a general class of queries on a shared-memory system for high throughputs.

(b) Online Sharing. To amortize computation costs, we develop a novel strategy that dynamically identifies *shared queries* whose computations substantially overlap with the computations performed by multiple queries in the batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all the queries in the batch. The shared subcomputations are essentially query evaluations for a small set of high degree source vertices, different from the source vertices of queries in the batch, such that they can be used to accelerate the evaluation of all queries in the batch.

Online sharing has multiple advantages over classical global indexing methods for optimizing evaluation of queries. First, indexing entails heavy weight precomputation used to build a large index that can be used to accelerate all future queries (e.g., Quegel [30] uses Hub-Accelerator based indexing). Second, as soon as the graph changes, precomputed indexing/profiling information becomes invalid. The *online sharing* as performed by SimGQ+ involves no precomputation and thus eliminates its high cost while also accommodating changes to the graph between different batches of queries. Thus, our approach applies to streaming/evolving graphs.

In SimGQ+, the evaluation of the batch of queries is carried out as follows. We partially evaluate the queries in the batch for a few iterations till some high degree vertices enter the frontiers of the queries in the batch. We *pause* the evaluation of the batch queries and select a small set of high degree vertices encountered. Treating selected vertices as source vertices of queries, we construct a batch of *shared queries* and then evaluate this batch. The results of shared queries are then used to *quickly update* the solutions of all queries in the original batch and hence accelerate their advance towards the final stable solution. Finally, we *resume* the paused evaluation of original batch till their stable solutions have been found. By simultaneously evaluating queries we also amortize the runtime overheads incurred, such as costs of accessing vertices and edges, synchronization costs, and maintaining frontiers as multiple queries traverse the same regions of the graph.

While conventional point-to-all queries have been widely studied, recent works on Quegel [30] and PnP [29] focus on point-to-point iterative graph queries (e.g., shortest path from a source vertex s to a destination vertex d), which is another type of important graph query. Both Quegel and PnP evaluate point-to-point queries one by one. In this work, we also extend SimGQ+ to efficiently evaluate a batch of point-to-point queries. In addition to batching, we propose query aggregation, an optimization which merges multiple point-to-point queries sharing the same source vertex into a coarse-grained one-to-many query with a single source vertex and multiple destination vertices. Similarly we may aggregate point-to-point queries sharing the same destination vertex into many-to-one queries. Query aggregation eliminates the shared computation among point-to-point queries and thus delivers better performance. In addition, we generalize pruning and direction prediction, two optimizations proposed in PnP, from the single point-to-point query scenario to the aggregated one-to-many query scenario to further improve performance of batched evaluation.

We implemented SimGQ+ by modifying the state-of-the-art Ligra [22] system. Our experiments with multiple input power-law graphs and multiple graph algorithms demonstrate best speedups ranging from 1.53× to 45.67× with batch sizes ranging up to 512 queries over the baseline implementation that evaluates the queries one by one using the state of the art Ligra system. Moreover, we show that both batching and sharing techniques contribute substantially to the speedups.

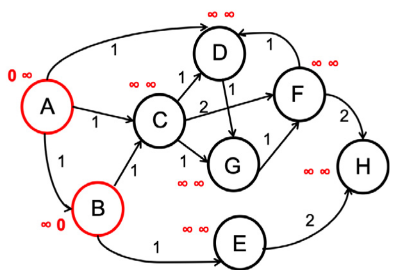
This paper makes the following contributions:

- A batching algorithm that amortizes the run-time overhead across a batch of iterative graph queries.
- A sharing algorithm and a heuristic for shared query selection to amortize the computation cost across a batch of point-to-all graph queries.
- A query aggregation algorithm and a direction prediction heuristic for efficient batched evaluation of point-to-point graph queries.
- A shared-memory implementation of three optimizations – batching, sharing, and query aggregation – as an extension of the Ligra framework, and thorough experiments that shows the performance gain from these optimizations.

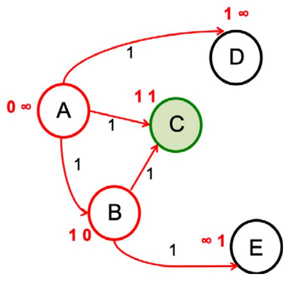
The remainder of the paper is organized as follows. In section 2 we first provide an overview of SimGQ+ and then present our algorithms in detail. In section 3 we present a batching strategy for evaluating point-to-point queries. In section 4 we evaluate SimGQ+. In section 5 we discuss related work. Finally, we conclude in section 6.

2. SimGQ+: evaluating a batch of queries

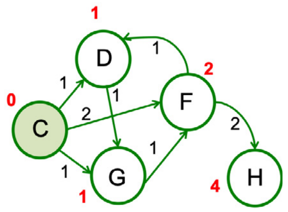
When a group of iterative graph queries are evaluated as a *batch*, following opportunities for speeding up their evaluation arise that are ignored when evaluating the queries one by one. First, it is easy to see that during batched evaluation, we can share the iteration overhead across the queries. This overhead includes the cost of iterating over the loop, synchronizing threads at the barrier, as well as fetching vertex values and edge lists of active vertices to update vertex values and the computation frontier. Second, synergy or overlap between computations performed by the queries can be exploited to reduce the overall computation performed. In particular, we can identify and evaluate *shared queries* whose results can be used to accelerate the evaluation of all the queries in the batch. The computation performed by the shared queries substantially represent subcomputations that are performed by many queries in the batch. This is because different queries typically traverse the majority of the graph and conse-



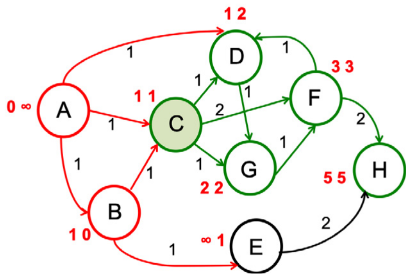
(a) *Initialization* for Batch(A,B) of SSSP Queries.



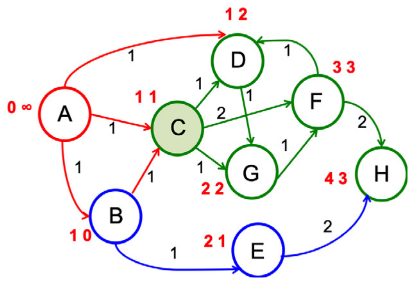
(b) *Phase I.* Evaluate Batch(A,B), Pause and Identify SSET.



(c) *Phase II.* Evaluate SSET.



(d) *Phase II Contd.* Update Batch(A,B) Results Using SSET Results.



(e) *Phase III.* Resume Batch(A,B) Evaluation till Termination.

Fig. 1. Overview of Sharing Among a Batch of Queries.

quently present an opportunity to share a subcomputation across multiple queries. By evaluating the shared queries once, we can speedup the evaluation of the entire batch of queries. Note that the shared queries must be identified dynamically because they may vary from one batch to another.

2.1. Overview of SimGQ+

Next we provide an overview of SimGQ+ via an illustrating example. Fig. 1 shows how a batch of two queries can be synergistically evaluated by identifying and evaluating shared queries first. While in the example we use a directed graph, our approach works equally well for undirected graphs with a minor adjustment. As in other works, each undirected edge is represented by a pair of directed edges with equal weight.

Initialization Step. Since all queries in a given batch are to be evaluated simultaneously, each vertex is assigned a vector to hold data values for all queries in the batch – each position in the vector corresponds to a specific query in the batch. In Fig. 1a we aim to solve a batch of two SSSP queries for source vertices A and B marked in red. Each node is annotated with a pair of initial values for the two queries, A first and then B. Initial value 0 is assigned to source vertices and value ∞ to all non-source vertices for each of the SSSP queries.

Phase I: Identifying Shared Queries. Simultaneously starting from the source vertices, we start traversing the graph updating the shortest path lengths for the processed vertices along the frontier as shown in red in Fig. 1b. The evaluation of the batch continues and once good candidate vertices for *shared queries* SSET are found, the evaluation of the batch is *paused*. Let us assume that after one iteration we identify SSSP(C) (C marked in green) as a good shared query candidate for the two queries in the batch in our example. Thus, we *pause* the evaluation of the batch queries and proceed to the next step to process the identified shared queries.

Phase II. Accelerating Batch Queries Using Shared Queries. In this step we evaluate the shared queries first, that is we evaluate them till their stable results have been computed. For example, in Fig. 1c we evaluate the shared query SSSP(C). Once the shared queries have been evaluated, their results are used to rapidly update the partial results of all the original batch queries as shown in Fig. 1d. Note that at this point the results of all vertices except B and E have already reached their final stable values. That is, the evaluation of batch queries has greatly advanced or *accelerated*.

Phase III. Completing the Evaluation of Batch Queries. In this final step we *resume* the evaluation of batch queries from the frontier at which the evaluation was paused earlier. In our example, the resumption of evaluation takes place at vertices B and E and finally the algorithm terminates after updating the results at vertices E and H. Note that if the acceleration performed in Phase II is effective, the combined cost of Phase I and Phase III would be significantly less than the cost of evaluating the batch without employing *sharing* affected via Phase II.

While the above example provides an overview of our approach, many algorithm details and heuristic criteria need to be developed. For example, there are different ways to select shared queries (queries on vertices with high centrality or high degree, queries on vertices that are reachable by most source vertices in the batch etc.). Since our work focuses on power-law graphs that have small diameter and skewed degree distribution, *high degree vertices* are the best candidates for global queries that in general traverse nearly the entire graph. Our algorithm first marks a set of high degree vertices as *potential shared vertices*. At runtime, a heuristic is used to select a small subset of *shared vertices* that are not only marked, but also have been encountered more frequently during partial evaluation of batch queries. After evaluating the *shared queries*, we use the results to quickly update the results of all batch queries. In subsequent subsections we present a push-style evaluation of a batch of queries assisted by our idea of using shared queries.

2.2. Push-style batch evaluation with sharing

Now we present a detailed algorithm that evaluates a batch of vertex queries, employing both batching and sharing, using Push model (a similar algorithm can be easily designed for the Pull model). In Algorithm 1, function EVALUATEBATCH (line 3) simultaneously evaluates a batch of vertex queries for source vertices $s_1, s_2, ..., s_k$, over a directed graph $G\ (V, E)$. The algorithm uses $M \subset V$ as a set of marked high degree vertices from which a small number of vertices are selected to form *shared queries*; different batches of queries yield different shared queries. In our experiments $|M|$ is set to 100 to provide choices that suit different batches, while up to 5 shared queries are selected to limit the overhead of sharing (i.e., SSET size is 5). The algorithm maintains an ACTIVE vertex set, the combined frontier for all queries in the batch. Although ACTIVE tells which vertices are active, it cannot tell which queries are associated with each active vertex. Therefore, in addition to ACTIVE, our algorithm maintains two fine-grained active lists, CURRTRACK and NEXTTRACK, to indicate for each active vertex all the queries whose frontier the active vertex belongs to. While CURRTRACK is the information for active set being processed, NEXTTRACK is the corresponding information for the active set being formed for the next super step of the algorithm. The RESULTT maintains the results of all the queries for each vertex, and at termination the results of all queries can be found in it.

Following the initialization step (lines 4-7), in each super iteration (lines 9-22), the vertices in ACTIVE vertex set are processed in parallel by calling function PROCESSBATCH (lines 25-37). This function updates the value of out-neighbors of active vertices in Push style fashion and generates NEWACTIVE containing the active vertices for next iteration which it returns to EVALUATEBATCH at the end. The work performed by the loop at line 9 executes the three phases of our algorithm. The first p iterations form Phase I, following which, next in Phase II first shared queries SSET are identified by calling SELECTSHAREDQS (line 15) and then the queries in SSET are evaluated (line 17). Finally, the evaluation of original batch of queries is accelerated by updating their results in RESULTT using the results of SSET queries in SHAREDT (line 19). Finally in Phase III the computation of batch queries is resumed and completed

Algorithm 1 Batched Evaluation With Sharing.

```
 1: Given: Directed graph $G(V, E)$; High Degree Set $M \subset V$ of Marked Vertices
 2: Goal: Evaluate a Batch of Queries, QUERYBATCH ← { $Q_1(s_1)$, $Q_2(s_2)$, ..., $Q_k(s_k)$ }

 3: function EVALUATEBATCH( QUERYBATCH )
 4:     ▷ [Initialization Step]
 5:     INITIALIZE RESULTT for QUERYBATCH
 6:     ACTIVE ← { $s_1, s_2, ..., s_k$ }; NEXTTRACK ← $\phi$; ITER ← 0
 7:     CURRTRACK ← { $(s_i, Q_i) : Q_i(s_i) \in$ QUERYBATCH }
 8:     ▷ Iterate till Convergence
 9:     while ACTIVE $\neq \phi$ do
10:         ▷ [Phase I: Iteration $\leq$ p] [Phase III: Iteration $>$ p]
11:         ▷ Process Active Vertices
12:         ACTIVE ← PROCESSBATCH ( ACTIVE, ITER, CURRTRACK, NEXTTRACK, RESULTT )
13:         if ITER = $p$ then ▷ [Phase II]
14:             ▷ Identify #SSET as the Most Frequently Visited
                    Vertices from $M$ as the source of Shared Queries
15:             SSET ← SELECTSHAREDQS ($M$, Visits, #SSET)
16:             ▷ Evaluate Shared Queries with Sources in SSET
17:             SHAREDT ← EVALUATEBATCH (SSET)
18:             ▷ Update RESULTT using SHAREDT
19:             SHAREUPDATEBATCH ( SSET, SHAREDT, RESULTT )
20:         end if
21:         CURRTRACK ← NEXTTRACK; NEXTTRACK ← $\phi$; ITER++
22:     end while
23:     return RESULTT
24: end function

25: function PROCESSBATCH ( ACTIVE, ITER, CURRTRACK, NEXTTRACK, RESULTT )
26:     NEWACTIVE ← $\phi$
27:     for all $v \in$ ACTIVE in parallel do
28:         for all $e \in G.outEdges(v)$ in parallel do
29:             ▷ Apply conventional Update on $e.dest$
30:             $changed$ ← EDGEFUNCBATCH ( $e$, CURRTRACK, NEXTTRACK, RESULTT)
31:             if ( ITER $\leq p$ ) and ( $e.dest \in M$ ) then Visits[$e.dest$]++ end if
32:             ▷ Update Active Vertex Set for next Iteration
33:             if $changed$ then NEWACTIVE ← NEWACTIVE $\cup$ {$e.dest$} end if
34:         end forall
35:     end forall
36:     return NEWACTIVE
37: end function
```

Algorithm 2 Batched Edge Update Function.

```
 1: function EDGEFUNCBATCH ($e$, CURRTRACK, NEXTTRACK, RESULTT)
 2:     ▷ Initialize RETVALUE to false.
 3:     ▷ Set to true if value of $e.dest$ is changed.
 4:     RETVALUE ← false
 5:     for all $Q_i(s_i) \in$ QueryBatch do
 6:         ▷ Only Attemp Update for Queries activated $e.source$
 7:         if ($e.source$, $Q_i$) $\in$ CURRTRACK then
 8:             ▷ Perform Update via $e$
 9:             if UPDATEFUNC($e$, $Q_i$, RESULTT) == true then
10:                 ▷ Schedule $e.dest$ for next Iteration
11:                 RETVALUE ← true
12:                 NEXTTRACK ← NEXTTRACK $\cup$ {($e.dest$, $Q_i$)}
13:             end if
14:         end if
15:     end for
16:     return RETVALUE
17: end function
```

in remaining iterations of the while loop. During Phase I the algorithm maintains a count of number of visits to each vertex in M (line 31). These counts are used for selecting vertices to form SSET, more visits implies greater relevance to queries in the original batch and hence higher priority for inclusion in SSET. Following the call to PROCESSBATCH in the p^{th} iteration (1^{st} in our experiments), we enter Phase II at which point SSET is built. The details of SSET construction are presented in Algorithm 3.

Function PROCESSBATCH loops over each outedge e of every active vertex, and calls function EDGEFUNCBATCH (Algorithm 2) to attempt update of $e.dest$ by relaxing edge e using conventional edge

Algorithm 3 Identify Shared Queries from M.

```
1: Given: High Degree Set $M \subset V$ of Marked Vertices
2:        Vector Visits: Number of Visits of All Vertices $\in M$
3:        Constant #SSET: # of Shared Vertices Selected
4: Goal: Select #SSET most frequently visited Vertices in $M$

5: function SELECTSHAREDQS ($M$, Visits, #SSET)
6:     ▷ Init: Set of Source Vertices for Shared Queries
7:     SSET ← $\phi$
8:     ▷ Init: Set of (vertex, vertex visits number) pairs
9:     VERTVISITSPAIRS ← $\phi$
10:    for all $v \in M$ do
11:        VERTVISITSPAIRS←VERTVISITSPAIRS∪{$v$,Visits[$v$]}
12:    end for
13:    ▷ Sort Vertices subject to Number of Visits
14:    Sort ( VERTVISITSPAIRS, moreVisits() )
15:    ▷ Select most frequently visited Marked Vertices
16:    for #SSET top {$v$, Visits[$v$]} ∈ VERTVISITSPAIRS do
17:        SSET ← SSET ∪ {$v$}
18:    end for
19:    return SSET
20: end function
```

Table 1
Conventional Updates for Five Algorithms.

ALG	RESULTT[s_i][$e.dest$] ← UPDATEFUNC (e, Q_i, RESULTT)
SSWP	CASMAX(RESULTT[s_i][$e.dest$], min(RESULTT[s_i][$e.src$], $e.w$)))
Viterbi	CASMAX(RESULTT[s_i][$e.dest$], RESULTT[s_i][$e.src$] / $e.w$)
BFS	CASMIN(RESULTT[s_i][$e.dest$], RESULTT[s_i][$e.src$] + 1)
SSSP	CASMIN(RESULTT[s_i][$e.dest$], RESULTT[s_i][$e.src$] + $e.w$)
TopkSSSP	KSMALLEST({RESULTT[s_i][$e.dest$]} ∪ {RESULTT[s_i][$e.src$] + $e.w$})

Table 2
Directed Graphs: SHAREUPDATEFUNC for Five Algorithms.

ALG	RESULTT[s_i][d] ← SHAREUPDATEFUNC(d,r,Q_i, SHAREDT,RESULTT)
SSWP	CASMAX(RESULTT[s_i][d], min(RESULTT[s_i][r], SHAREDT[r][d]))
Viterbi	CASMAX(RESULTT[s_i][d], RESULTT[s_i][r] * SHAREDT[r][d])
BFS	CASMIN(RESULTT[s_i][d], RESULTT[s_i][r] + SHAREDT[r][d])
SSSP	CASMIN(RESULTT[s_i][d], RESULTT[s_i][r] + SHAREDT[r][d])
TopkSSSP	KSMALLEST({RESULTT[s_i][$e.dest$]} ∪ {RESULTT[s_i][r] + SHAREDT[r][d]})

update function UPDATEFUNC. If the relaxation is successful, i.e. the value of $e.dest$ is changed, $e.dest$ becomes an active vertex for next iteration. Note that function EDGEFUNCBATCH does not blindly relax e for all queries. Instead it looks up CURRTRACK to check which queries activated $e.source$ in the previous iteration, and only attempts update of value of $e.dest$ for corresponding queries.

If lines 13-20 are eliminated, the algorithm will not perform sharing and thus its execution will revert to simple batched evaluation. The conventional edge update function UPDATEFUNC for four algorithms is given in Table 1. Here CASMIN(a, b) sets $a = b$ if $b < a$ atomically using compare-and-swap); and CASMAX(a, b) sets $a = b$ if $b > a$ atomically using compare-and-swap).

Finally, Algorithm 4 shows how we accelerate the convergence of the solution of the original batch of queries in RESULTT using the results of the shared queries in SHAREDT. Since the cost for looping over all vertices and applying share updates is significant, we limit the number of shared vertices with which each query is used to speed up convergence of property values by choosing a small SSET size. Let us see how the result of a shared query with source vertex r can benefit a batch query suppose the reachability is known to be true. Given a vertex d, its value in query Q_i can take advantage of the shared result of subquery on vertex r in SHAREDT as follows: SHAREUPDATEFUNC(d, r, Q_i, SHAREDT, RESULTT). The above function for four benchmarks is given in Table 2. For example, for SSSP,

$$\text{RESULTT}[s_i][r] + \text{SHAREDT}[r][d]$$

Algorithm 4 Accelerate Batch Queries Using Shared Queries From SSET.

```
1: function SHAREUPDATEBATCH (SSET, SHAREDT, RESULTT)
2:     for all $Q_i(s_i) \in$ QUERYBATCH do
3:         for $r \in$ SSET do
4:             ▷ Update using $r$ only if $r$ is reachable from $s_i$
5:             if RESULTT[$s_i$][$r$] $\neq -1$ then
6:                 for $d \in$ ALLVERTICES do
7:                     ▷ Attempt Update if $d$ is reachable from $r$
8:                     if SHAREDT[$r$][$d$] $\neq -1$ then
9:                         ▷ Update $d$ for Query $i$ using $r$
10:                        SHAREUPDATEFUNC ( $d$, $r$, $Q_i$, SHAREDT, RESULTT )
11:                    end if
12:                end for
13:            end if
14:        end for
15:    end for
16: end function
```

is a safe approximation of the shortest path value from source vertex of q_i to d via r, and we can use the estimation to accelerate the convergence of the value of d.

For undirected graphs, when applying update using result of shared queries, we can benefit from a more accurate measurement of the property value from source vertex to shared vertex. Take SSSP as an example. Given an undirected graph, SHAREDT[r][s_i] can be used as the accurate measurement of the distance from s_i to r. Compared with RESULTT[s_i][r] used in Table 2, which is an approximation value, SHAREDT[r][s_i] can be used to compute a better estimation of the distance between s_i and d and therefore give better acceleration on the evaluation of original batch queries.

2.3. Applicability

Our sharing algorithm can be applied to batched iterative graph algorithms where each query in the batch begins at single source vertex and the property values from these sources to all other vertices are computed. Sharing of results of subqueries is effective because they represent overlapping subcomputations. Graph problems with dynamic programming solutions have the opportunity to benefit from our sharing algorithm because of the optimal substructure property of dynamic programming. Examples include monotonic computations like SSWP, Viterbi, TopkSSSP, and BFS used in our evaluation as well as other non-monotonic algorithms like Personalized Page Rank (PPR) [6] used by recommender services like twitter and Single-Source SimRank (SimRank) [7] queries that are evaluated to compute similarities of graph nodes. It does not apply to algorithms with a global solution, i.e. not originating at source-vertex (e.g., Connected Components). Sharing will work less effectively for local queries like 2-Hop queries due to low overlap between them; however, local queries are inexpensive and can be processed efficiently with batching alone. Sharing works well on power-law graphs as they contain high centrality nodes but it is less effective for high-diameter graphs like road-networks. Only when source vertices are in proximity of each other can there be significant reuse in high-diameter graphs.

3. PTP in SimGQ+: evaluation of a batch of point-to-point queries

In the previous section, we have discussed how to optimize the batched evaluation of a group of one-to-all graph queries. In addition to one-to-all queries, point-to-point query (e.g., a query that computes the shortest path distance between a pair of vertices) is another important category of graph queries which has many applications such as road routing and network analysis [30]. While prior works [30,29] are focused on accelerating a single point-to-point query, typically the use scenarios require multiple queries to be evaluated for a given analytic task. In this section, we pro-

Table 3
Relationship between Execution Time and Number of Queries.

Aggregation	Point-To-Point	One-To-Many	Many-To-One
# of Queries	50	38	13
Time w/o Batching	21.83s	16.90s	5.58s
Time w/ Batching	4.62s	3.89s	1.66s

pose an algorithm that efficiently evaluates multiple point-to-point queries.

3.1. *Query aggregation - exploit shared computation*

When a group of point-to-point queries are evaluated simultaneously, new opportunities for optimization arise. First of all, we may apply batching for better resource utilization and cache locality, just like what we did for point-to-all queries in the previous section. Moreover, we can develop optimizations specific to multiple point-to-point queries. In this work, we are particularly interested in eliminating shared computations across batch queries. We consider *forward-aggregation* which combines multiple point-to-point queries that share the same source vertex into one point-to-many query which has one source vertex and multiple destination vertex. Similarly we consider *backward-aggregation* which combines the point-to-point queries that share the same destination vertex into a many-to-point query with one destination vertex and multiple source vertices. Such query aggregation reduces the number of queries and can further reduce the computation overhead because it reduces the number of distinct frontiers we need to maintain.

To illustrate the performance benefit from query aggregation, we conducted the following motivating experiment. Fifty point-to-point shortest path queries are selected such that many of them share source vertex or destination vertex. The input graph is soc-LiveJournal [12] with 4847571 vertices and 68993773 edges. In Table 3, we compare the total running times under three different aggregation policies. Point-to-Point is the baseline without aggregation. One-to-many is the result of forward aggregation. Many-to-one is the result of backward aggregation. As we can see, both one-to-many and many-to-one generate fewer queries than point-to-point and also run faster than point-to-point which shows that query aggregation is effective and leads to less query time. In addition, by comparing the running time with and without batching, we observe that batching is effective for aggregated queries. Thus batching and sharing can be applied together for better performance.

3.2. *Adapt pruning to multiple one-to-many query scenario*

In our prior work PnP [29], we introduced an online pruning optimization to accelerate point-to-point iterative graph algorithms by eliminating the wasteful propagation which are determined not to contribute to the final answer. Pruning is achieved by comparing the new value of an active vertex v (i.e., $v.value$) with the current value of the destination vertex d (i.e., $d.value$). For instance, in the case of the shortest path algorithm, we can safely prune out vertex v from the active vertex frontier if $v.value \geq d.value$. Pruning reduces the amount of vertex propagation and leads to early termination compared with standard iterative graph algorithms.

We can adapt pruning from the scenario of a single point-to-point query to the scenario with a single one-to-many query. Suppose we have an one-to-many query with a single source vertex s and k destination vertices d_1, d_2, ..., d_k. We can safely prune out vertex v from the active vertex frontier if $v.value$ cannot contribute to any of the k point-to-point queries which include $s \rightarrow d_1$, $s \rightarrow d_2$, ..., $s \rightarrow d_k$. In the case of the shortest path problem, we can prune out v if $v.value \geq max(d_i.value)$ where $i \geq 1$ and $i \leq k$.

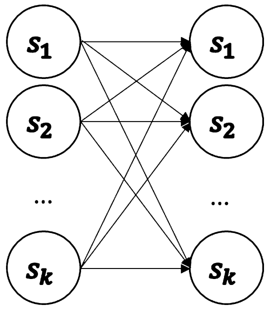


Fig. 2. Full-Mapping Queries.

We can further generalize pruning from the scenario with a single one-to-many query to the scenario with multiple concurrent one-to-many queries. Let us see how pruning can be combined with batching. Suppose we have l one-to-many queries which originate from source vertices $s_1, s_2, \ldots, s_l$. Each one-to-many query has k destinations. Let Q_i denote the ith one-to-many query which originates from s_i. Let $\textsc{Threshold}[Q_i]$ denote the pruning threshold for query Q_i. Then $\textsc{Threshold}[Q_i] = \max(\textsc{ResultT}[Q_i][d_j])$ for all $j \in [1, k]$ where $\textsc{ResultT}[Q_i][d_j]$ is the tentative shortest path value from s_i to d_j. The pruning condition for vertex v in the multiple query scenario will be $v.value >= max(\textsc{Threshold}[Q_i])$ for $i \in [1, l]$.

3.3. *Discussion of full-mapping workload - breaking tie between forward and backward aggregation using direction prediction*

Yan et al. [30] observed that many applications on large graphs simply require computing point-to-point variants of heavyweight computations. As an example, when analyzing a graph that represents online shopping history of shoppers, a business may be interested in all point-to-point queries over an important set of shoppers. Therefore in this work we focus on this kind of full-mapping workload. Moreover, we identify high degree vertices in the input graph as vertices of interest because high degree vertices are usually important vertices in power-law graphs.

Under this workload, the set of input point-to-point queries can be represented as a full mapping from a set of source vertices to a set of destination vertices (see Fig. 2). As a result, forward/backward aggregation generate a minimal number of one-to-many/many-to-one queries. In other words, there is a tie between the forward and backward aggregation. Then the question comes which direction to be chosen for aggregation: forward or backward?

Inspired by the direction prediction heuristic developed in PnP [29], we developed a heuristic for predicting the faster direction for a batch of point-to-point queries based on the estimation of work. While in PnP direction prediction is conducted dynamically after a bidirectional phase one execution by comparing the frontier sizes from different directions, in this work we predict the direction for a given batch of point-to-point queries statically before iterative computation starts. We make this design choice because we observe that the computation for point-to-point queries between high-degree vertices usually finishes in a few iterations making the dynamic direction selection overhead high relative to the small overall work.

Here is the direction prediction heuristic that we employ. For each point-to-point query in the input batch, we assume the query is forward-fast if the outdegree of the source vertex is greater than the indegree of the destination vertex and similarly assume the query is backward-fast otherwise. If there are more forward-fast queries in the batch, we predict the forward direction is faster and run the whole batch in the forward direction starting from the source vertices over the original graph. Otherwise we predict the backward direction to be the faster direction, and run the whole batch in the backward direction starting from the destination vertices over the edge-reversed graph.

An alternative direction prediction heuristic for a batch is to compare the sum of outdegrees of all source vertices and the sum of the indegrees of all destination vertices and select the direction with smaller total degree. However, we found that this metric does not reflect the relative amount of work in different directions as accurate as the first approach because this alternative approach may over-emphasize the importance of a single point-to-point query in the batch and consequently misleadingly hide the impact of other batch queries. For instance, among 16 point-to-point queries in a batch, 15 queries are forward-fast and only 1 query is backward-fast; however the backward-fast query has a source with very high outdegree and a destination with very small indegree and dominate the overall prediction.

3.4. Push style batched evaluation of full-mapping point-to-point queries

Now we present a detailed algorithm that computes the pairwise property values between each pair of vertices from a set of important vertices using the Push model. In Algorithm 5, function EVALUATEPAIRWISE works as follows. First, function PREDICT (line 5) applies one of the static degree-based heuristics (discussed in Section 3.3) to predict the faster direction for the given input query on the given input graph. For k query vertices, there are k^2 corresponding point-to-point queries. Based on the prediction result, the algorithm decides how to aggregate these point-to-point queries into more coarse-grained queries. If the predicted direction is forward, k^2 point-to-point queries are aggregated in the forward direction into k one-to-many queries each of which has a single source vertex and k destination vertices (line 10-13). If the predicted direction is backward, point-to-point queries are aggregated in the backward direction into k many-to-one queries each of which has a single destination vertex and k source vertices (line 15-18). After generating the aggregated queries, function EVALUATEBATCHEDONETOMANY is called to evaluate the aggregated queries as a batch (lines 13 and 18). To benefit from a unified interface for evaluating one-to-many queries and many-to-one queries, the evaluation of many-to-one queries is conducted as follows. We first convert many-to-one queries into one-to-many queries by calling the function REVERSE (e.g., many-to-one query $(\{s_1, s_2, ..., s_k\}, s_i)$ is reversed to an one-to-many $(s_i, \{s_1, s_2, \ldots, s_k\})$). After the conversion, we can get the result of the original many-to-one query by evaluating the reversed one-to-many queries on the edge-reversed graph $\hat{G}$ (line 18).

Let us dive into function EVALUATEBATCHEDONETOMANY which simultaneously evaluates a batch of one-to-many queries over a directed graph $G(V, E)$ where the *ith* query originates at source vertex s_i and has k destination vertices $s_1, s_2, ..., s_k$. The function is explained in Algorithm 6. It is similar to the simple batching algorithm for point-to-all queries (Algorithm 1 without lines 13-20). The algorithm maintains an ACTIVE vertex set, the combined frontier for all queries in the batch as well as two fine-grained active lists, CURRTRACK and NEXTTRACK, that provides information about which vertex is activated by which query. The RESULTT maintains the results of all the queries for each vertex. What makes a difference is pruning (line 18). After each iteration of propagation, THRESHOLD[Q_i] for pruning is updated for each one-to-many query Q_i from QueryBatch using the tentative query results by applying an aggregation for loose boundary (line 14-16). For instance, AGGREGATELOOSE is *max* in the case of shortest path and BFS while it

Algorithm 5 Pairwise Evaluation between a Group of Vertices.

```
 1: Given: Directed Graph: $G(V, E)$; The Edge-Reverse Graph: $\hat{G}(V, \hat{E})$; and Vertex
    set QueryVertices which contains a group of $k$ vertices
 2: Goal: Compute point-to-point values between each pair of vertices in
    QueryVertices

 3: function EVALUATEPAIRWISE(QueryVertices($s_1, s_2, s_3, ..s_k$), $G$, $\hat{G}$)
 4:     ▷ Predict faster Direction
 5:     Prediction ← PREDICT(QueryVertices, $G$)
 6:     ▷ Generate Point-to-Point Queries from Input Vertex Set
 7:     PTPQueries ← GENPTPQUERIES(QueryVertices);
 8:     ▷ Aggregate Point-To-Point Queries and Evaluate in the Predicted Direction
 9:     if Prediction = Forward then
10:         ▷ Aggregate Point-To-Point Queries into One-To-Many Queries
11:         OneToManyQueries ← FWDAGGREGATE(PTPQueries)
12:         ▷ Evaluate aggregated Queries
13:         EVALUATEBATCHEDONETOMANY(OneToManyQueries, $G$)
14:     else
15:         ▷ Aggregate Point-To-Point Queries into Many-To-One Queries
16:         ManyToOneQueries ← BWDAGGREGATE(PTPQueries)
17:         ▷ Evaluate aggregated Queries
18:         EVALUATEBATCHEDONETOMANY(REVERSE(ManyToOneQueries), $\hat{G}$)
19:     end if
20: end function
```

Algorithm 6 Batched Evaluation of One-To-Many Queries.

```
 1: Given: Directed Graph Graph(V, E); QueryBatch which is set of k
    One-To-Many queries: $Q_1(s_1, \{s_1, s_2, s_3, ..s_k\})$, $Q_2(s_2, \{s_1, s_2, s_3, ..s_k\})$,
    $\cdots Q_k(s_k, \{s_1, s_2, s_3, ..s_k\})$
 2: Goal: Evaluate the given batch of One-To-Many queries

 3: function EVALUATEBATCHEDONETOMANY( QueryBatch )
 4:     ▷ Initialization Step
 5:     Initialize RESULTT for QueryBatch
 6:     ACTIVE ← { $s_1, s_2, ..., s_k$ }; NEXTTRACK ← $\phi$;
 7:     CURRTRACK ← { $(s_i, Q_i) : Q_i(s_i) \in$ QueryBatch }
 8:     Initialize THRESHOLD[] for pruning for each $Q_i$ in QueryBatch
 9:     ▷ Iterate till Convergence
10:     while ACTIVE $\neq \phi$ do
11:         ▷ Process Active Vertices
12:         ACTIVE ← PROCESSBATCH (ACTIVE, CURRTRACK,NEXTTRACK, RESULTT)
13:         ▷ Update the Pruning Threshold for each One-To-Many Query in the
            Batch
14:         for all $(s_i, \{s_1, s_2, ..s_k\}) \in$ QueryBatch in parallel do
15:             THRESHOLD[$Q_i$] ← AGGREGATELOOSE(ResultT[$Q_i$][$s_j$]) for $j = 1..k$
16:         end forall
17:         ▷ Prune active frontier using pruning threshold
18:         ACTIVE ← PRUNEACTIVE(ACTIVE, NEXTTRACK, THRESHOLD, RESULTT)
19:         CURRTRACK ← NEXTTRACK; NEXTTRACK ← $\phi$;
20:     end while
21:     return RESULTT
22: end function
```

is *min* in the case of widest path and Viterbi algorithm. Pruning threshold is computed as loose boundary rather than tight boundary because we cannot prune a vertex v for a one-to-many query as long as the new value of v may contribute to the value of any (rather than all) destination vertices. With the updated THRESHOLD, the active vertex frontier is pruned by calling the function PRUNEACTIVE (line 18). Function PRUNEACTIVE is described in detail in Algorithm 7. An active vertex v cannot be pruned from the active vertex set if at least one of the queries does not want to prune it (line 6-7). Function DONOTPRUNE (line 6) varies from benchmark to benchmark. For instance, in the case of shortest path problem, DONOTPRUNE returns true if and only if RESULTT[Q_i][v] $<$ THRESHOLD[Q_i].

4. Experimental evaluation

4.1. Evaluation of SimGQ+

4.1.1. Experimental setup for SimGQ+

For evaluation we implemented our SimGQ+ framework using Ligra [22] which uses the Bulk Synchronous Model [25] and pro-

Algorithm 7 Prune Active Vertex Frontier using Threshold.

```
 1: function PRUNEACTIVE( ACTIVE, NEXTTRACK, THRESHOLD, RESULTT)
 2:   NEWACTIVE ← $\phi$
 3:   for all $v \in$ ACTIVE in parallel do
 4:     for all $Q_i \in$ QueryBatch do in parallel
 5:       ▷ $v$ cannot be pruned if any query needs propagation of new value of $v$
 6:       if DONOTPRUNE($v$, $Q_i$) then
 7:         NEWACTIVE ← NEWACTIVE ∪ {v}
 8:       else
 9:         NEXTTRACK ← NEXTTRACK \ {($v$, $Q_i$)}
10:       end if
11:     end forall
12:   end forall
13:   return NEWACTIVE
14: end function
```

Table 4
Input graphs used in experiments.

Graphs	#Edges	#Vertices
Twitter (TT) [3]	2.0B	52.6M
Twitter (TTW) [9]	1.5B	41.7M
LiveJournal (LJ) [2]	69M	4.8M
PokeC (PK) [23]	31M	1.6M

Table 5
BASELINE – Total Execution Times for Evaluating Randomly Selected Queries One by One in Seconds on the Ligra [22] System. For first 3 benchmarks 512 queries are used and for TopkSSSP we use 64 queries.

Graph	SSWP	Viterbi	BFS	Top 2 & 1 SSSP	
TTW	2,989s	3,737s	2,574s	4,073s	2,337s
TT	3,949s	4,902s	3,538s	2,768s	1,574s
LJ	134s	258s	102s	389s	226s
PK	63s	116s	55s	232s	123s

vides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. We evaluate our techniques for evaluation of batches of queries using four benchmark applications (SSWP – Single Source Widest Path, Viterbi [11], BFS – Breadth First Search, and TopkSSSP – Top k Single Source Shortest Paths). We used four real world power-law graphs shown in Table 4 in these experiments – TT [3] and TTW [9] are large graphs with 2.0 and 1.5 billion edges respectively; and LJ [2] and PK [23] are smaller graphs with 69 and 31 million edges respectively. Benchmarks are implemented using the PUSH model on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 512 GB memory, and running CentOS Linux 7.

For each combination of benchmark application and input graph, we used 512 randomly generated queries to carry out the evaluation, except for TopkSSSP for which we use 64 queries because of runtime cost. The *baseline* total execution times when the queries are evaluated one by one is given in Table 5. Because TTW and TT are far bigger in size than LJ and PK, the execution times for TT and TTW are higher.

4.1.2. Benefits of sharing and batching in SimGQ+

In this section we present the results of our algorithm, we refer to them as Batch+Share. In addition, we also collect execution times of algorithm that uses batching but no sharing, we refer to this algorithm as Batch. Since the batch size is an important parameter in this evaluation, we vary batch sizes from 4 queries (the smallest) to a very large number of 512 queries. For TTW and TT the maximum batch size was limited to 256 because our machine did not have sufficient memory to run 512 queries for very large graphs. For TopkSSSP maximum batch size is limited to 64 due to its high runtimes.

The results of running the above algorithms are presented in Table 6 and Fig. 3. While Table 6 presents the total execution times for 512 queries for batch sizes (number in parentheses) that yielded the highest speedup for each of the algorithms, Fig. 3 presents average per query execution times for all batch sizes for TT the largest graph.

The data in this Table 6 shows that our algorithms yield speedups of up to 45.67× over the baseline that executes the queries one by one using the state of the art Ligra system. For the first two benchmarks of SSWP and Viterbi the Batch+Share algorithm delivers speedups ranging from 22.11× to 45.67×. In contrast, for the last two benchmarks of BFS and TopkSSSP the highest speedups observed range from 1.53× to 6.63×.

The sharing algorithm is more profitable if the result values of queries fall in a narrow range and hence often overlap. Like the result of SSWP query is usually an integer between 17 and 25, and the answer of Viterbi is between 0 and 1. In these cases, sharing produces lots of stable values and reduces the number of iterations because vertices made stable by sharing will never be activated again. Sharing is also effective when the vertex update function is expensive even if it produces few stable values – TopkSSSP is a representative graph algorithm from this category. Here sharing reduces the number of updates by 34% but produces few stable values. BFS does not fall into any of these two categories and thus, as expected, does not benefit much from sharing.

Let us consider results in Figs. 3 that present the *average per query execution times* for varying batch sizes. The trends for the first three benchmarks show that performance continues to improve with increasing batch sizes. For Batch the improvement is due to greater amortization of runtime overheads while for Batch+Share the improvement is greater due to additional benefits of sharing. Further, we observe that on our machine, once we cross the batch size of 64, the improvements in performance are relatively small although the best performances reported in Table 6 are for batch sizes of 256 and 512 for majority of the cases (i.e., different graphs and benchmarks). Based upon the trends observed in Fig. 3, for a larger machine with more memory and number of cores, performance can be expected to scale further with batch size. For TopkSSSP while there is less variation with batch size the difference between Batch and Batch+Share is substantial.

4.1.3. Contributions of sharing vs. batching in SimGQ+

We observed that for SSWP and Viterbi both sharing and batching are responsible for delivering high performance while for TopkSSSP batching does not provide benefit, and for BFS sharing does not deliver additional performance improvement. We analyze the cost and benefit of sharing to show that for first three benchmarks the benefit far outweighs the cost while for BFS the benefit is smaller than the cost incurred.

Using the execution times of Batch, which is essentially a shared-memory version of MultiLyra, as baseline, Table 7 presents the speedups achieved by Batch+Share. As we can see from the results, for benchmarks of SSWP and Viterbi, the speedups range from 5.14× to 12.64× demonstrating that sharing delivers substantial additional speedups over batching alone for these benchmarks. For benchmark of TopkSSSP, the benefit from sharing is less, but there are still descent speedups of up to 2.16× due to sharing. On the other hand, for benchmark of BFS there is even some slowdown.

The Cost and Benefit of sharing are also shown explaining the above results. The Cost is the time spent in Phase II while Benefit is reduction in total time spent on Phase I + Phase III due to sharing based updates performed by Phase II. Both the Cost and Benefit are presented as fraction of execution times of corresponding Batch algorithms. Thus, the Speedups are related to the Cost and Benefit as follows: $Speedup = 1/(1 + Cost - Benefit)$. For SSWP, Viterbi,

Table 6
Best Batching+Sharing and Batching Execution Times in `Seconds` for all Queries and Corresponding (Batch Sizes) and Speedup Over No-Batching Baseline times from Table 5.

Algorithm	SSWP (512 Queries)			Viterbi (512 Queries)			BFS (512 Queries)		
	TTW								
Batch+Share	71	(256)	42.37×	86	(256)	43.42×	440	(128)	5.84×
Batch	629	(256)	4.75×	729	(256)	5.13×	388	(256)	6.63×
	TT								
Batch+Share	90	(256)	43.96×	107	(256)	45.67×	723	(128)	4.90×
Batch	1034	(64)	3.82×	1274	(64)	3.85×	692	(128)	5.12×
	LJ								
Batch+Share	6	(512)	22.11×	12	(128)	22.27×	23	(256)	4.36×
Batch	37	(256)	3.63×	59	(256)	4.34×	18	(256)	5.63×
	PK								
Batch+Share	2	(512)	28.38×	4	(128)	28.97×	11	(512)	5.01×
Batch	20	(512)	3.24×	30	(256)	3.89×	9	(512)	6.40×

Algorithm	Top 2 & 1 SSSP (64 Queries)						
	TTW						
Batch+Share	2671		1260	(32)	(32)	1.53×	1.86×
Batch	3652	1876	(32)	(8)	1.12×	1.25×	
	TT						
Batch+Share	1605		858	(8)	(8)	1.73×	1.84×
Batch	2768		1574	(1)	(1)	1.00×	1.00×
	LJ						
Batch+Share	237		135	(64)	(64)	1.64×	1.67×
Batch	375		190	(32)	(32)	1.04×	1.19×
	PK						
Batch+Share	119		58	(64)	(64)	1.95×	2.13×
Batch	196		98	(16)	(16)	1.19×	1.26×

and TopkSSSP, the Benefit far exceeds the Cost while for BFS, the Cost exceeds the Benefit hence the observed speedup results. Finally, Table 8 summarizes how the overall speedups achieved for SSWP and Viterbi can be *factored* between batching and sharing showing the importance of employing both batching and sharing techniques.

The cost of sharing is reasonable because overheads of sharing come from three sources and all of them are low. First, we need to maintain a counter of the number of visits for each marked high degree vertex in Phase I. This overhead is negligible because we only mark a very small amount of high degree vertices (e.g., 100 out of millions in the current setting) and Phase I is very short (e.g., 1 iteration) and thus has relatively small frontier sizes. Second, we need to solve the shared queries in Phase II. Given that it only computes a small number of shared queries (e.g., only 5 from 100) while the batch size for original queries can be much larger (up to 512), the cost is amortized well across all queries in a batch and thus it has little impact on each individual query. Third, we introduce extra computation cost when applying the result of shared queries to accelerate the convergence of original query. Since this step is a linear scan of the array, it leads to better cache performance due to spatial locality compared with the usual updates for a query which can be randomly scattered across the value array in Ligra. Besides, our sharing algorithm only allows each query to reuse the result of one shared query and only once, keeping the reuse cost low.

To better understand the effectiveness of sharing, we also collected the *stable value* percentages – this is the percentage of vertices reachable from the source vertex whose vertex values converge as a result of performing share updates. We collected this data for the Batch+Share configuration. Since we pause the original computation only after the first iteration (i.e., $p = 1$), the percentage of vertices that are stable prior to sharing updates is negligible (less than 0.01%). The percentages of values that are stable following sharing updates are presented in Table 9. As shown in the table, sharing greatly benefits SSWP and Viterbi as it causes nearly all the values ($> 99\%$) to converge. To explain the phenomenon that Top2SSSP and Top1SSSP has lower stable percentage than BFS but sharing delivers much more speedups for the former than the latter, we collected the reduction in number of vertex updates resulting from sharing. It turns out that the reduction for Top2SSSP and Top1SSSP (34%) is much higher than the reduction for BFS (7%).

4.1.4. Sensitivity of SimGQ+ performance to the **p** value

All our preceding experiments were performed for **p** value of **1**, i.e. Phase I lasted one iteration following which Phase II was performed and then the updates from Phase II results optimized the remainder of time spent in Phase III till convergence. We varied the **p** value from 1 to 3 and compared the speedups that were obtained by sharing over batching alone. The results in Table 10 show that **p** value 1 delivers best overall speedups and the trend is that speedup falls as **p** value is increased. The only exceptions are LJ::Viterbi and PK::Viterbi where **p** value of 2 slightly outperforms **p** value of 1 (5.48× v.s. 5.14×, 8.57× v.s. 8.22×). There is a performance tradeoff in selecting **p** value. A smaller **p** enables an earlier reuse which leads to earlier convergence of queries. However, if **p** is small, limited number of marked high degree vertices may be visited and considered as candidates for sharing. We conclude the following from this experiment. First, executing Phase I for one iteration is sufficient as high quality SSET nodes have already been encountered. Second, executing Phase II early has the added benefit that greater fraction of overall iterations is optimized by the updates performed from the results of Phase II. We observe that **p** value of 1 causes sharing to deliver much higher speedups than **p** value of 2 for SSWP and Viterbi on large graphs than small

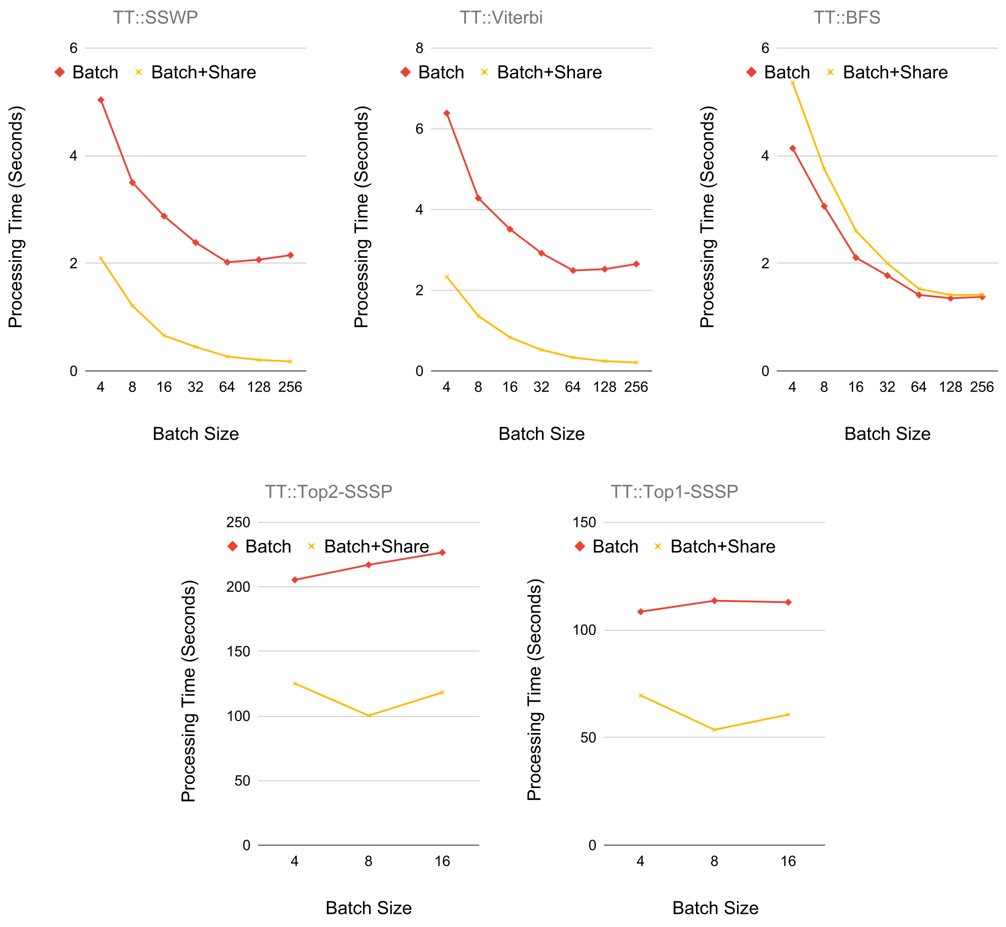


Fig. 3. Average Per Query Execution Times of Batch vs. Batch+Share.

graphs. For example, for the TT graph on Viterbi benchmark, the speedup over batching alone for **p** value of 1 is 12.64× while for the second best **p** value of 2, is much smaller 6.36×.

4.1.5. *Dynamic selection of SSET in SimGQ+*

One of the key characteristics of our algorithm is that the vertices in SSET are selected dynamically during the evaluation of a batch of queries. This has two main advantages. First, the selection of SSET vertices is customized to the batch of queries being evaluated. This is important that different batches may contain queries that are close to, in terms of number of hops, different high degree vertices and selection of closer high degree vertices offers greater opportunities of sharing. Second, our technique can be used to speedup the evaluation even when only a single batch of queries is to be evaluated. Note that alternatively techniques can be devised to profile executions of batches to identify SSET vertices and then use them to implement sharing in future batches. However, such an approach would lose both of the advantages of our approach mentioned above.

We next confirm that dynamic custom selection of SSET vertices for each batch does indeed lead to selection of different high degree vertices which deliver better speedups. We performed an experiment in which we split 256 queries for the two large graphs TTW and TT into four batches of 64 queries each. We identified the SSET vertices using the first batch and used it to perform sharing in the other three batches. Table 11 presents batch running time as follows: time using a single dynamically selected SSET vertex for the batch $\rightarrow$ time using a single dynamically selected SSET vertex in the first batch. The results show that for TTW::SSWP, TTW::Viterbi, and TT::SSWP custom/dynamic selection of SSET vertices for the last three batches delivers better performance (i.e., lower execution times) than the speedups that result from using SSET vertices identified using the first batch. For TTW::Viterbi batches 1 and 4 selected the same vertex and hence there is no change in execution time. For TT::Viterbi the nodes selected give nearly the same performance.

Finally, we examined the identities of selected SSET vertices for various batches to study the diversity of SSET vertices. In Table 12 we present actual number of distinct vertices included in SSETs versus the minimum number (size of SSET) and maximum number (number of batches × the size of SSET) of distinct vertices that can be observed. We found that the number of distinct SSET vertices selected are well above the minimum, i.e. during evaluation of different batches often different vertices are selected as SSET vertices.

4.2. *Evaluation of point-to-point queries in SimGQ+*

4.2.1. *Experimental setup*

For evaluating a batch of point-to-point (PTP) queries, we reused the batching interface from SimGQ and on top of that we implemented optimizations for batched processing of point-to-point queries including query aggregation, dynamic pruning, and direction prediction. We evaluated our techniques using four benchmark applications – SSWP, Viterbi, BFS, and SSSP. We once again used as input the four power-law graphs in Table 4. Benchmarks are implemented using the PUSH model. Experiments are conducted on a machine with 32 cores and 512 GB memory as described in Section 4.1.1.

Table 7
Batch+Share Over Batch Alone: Cost of Phase II, Benefit of Phase II, Speedup Due to Batch+Share Over Batch Alone. Speedups computed for best Batch+Share configurations for all Queries.

SSWP (512 queries)			Viterbi (512 Queries)			BFS (512 Queries)		
TTW								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.08	0.97	8.92×	0.09	0.97	8.47×	0.15	0.07	0.93×
TT								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.06	0.98	12.26×	0.06	0.98	12.64×	0.11	0.07	0.96×
LJ								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.12	0.96	6.43×	0.12	0.92	5.14×	0.26	-0.03	0.77×
PK								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.09	0.98	8.76×	0.08	0.96	8.22×	0.20	-0.08	0.78×

Top 2 & 1 SSSP (64 Queries)					
TTW					
Cost		Benefit		Speedup	
0.04	0.04	0.31	0.41	1.37×	1.60×
TT					
Cost		Benefit		Speedup	
0.09	0.09	0.63	0.62	2.16×	2.12×
	LJ				
Cost		Benefit		Speedup	
0.02	0.02	0.43	0.39	1.69×	1.58×
PK					
Cost		Benefit		Speedup	
0.02	0.02	0.50	0.50	1.94×	1.94×

Table 8
Factoring Speedups: Batching × Sharing = Total Speedup.

SSWP	Viterbi
TTW	
4.75 × 8.92 = 42.37×	5.13 × 8.47 = 43.42×
TT	
3.58 × 12.26 = 43.96×	3.61 × 12.64 = 45.67×
LJ	
3.44 × 6.43 = 22.11×	4.33 × 5.14 = 22.27×
PK	
3.24 × 8.76 = 28.38×	3.52 × 8.22 = 28.97×

For each combination of benchmark application and input graph, we evaluate the point-to-point property values between each pair of vertices from a set of query vertices which are the vertices with highest total degrees from the input graph and with both indegree and outdegree above a set default threshold of 500.

Since number of query vertices is an important parameter in this evaluation, in the experiments we vary this number from 4 vertices (i.e., 16 point-to-point queries) to 128 vertices (i.e., 16,384 point-to-point queries). The maximum number of query vertices is set to 64 (i.e., 4,096 point-to-point queries) for TTW graph and 32 (i.e., 1,024 point-to-point queries) for TT graph because of high execution times due to large sizes of these graphs.

4.2.2. *Effectiveness of aggregation and batching*

In this section we present the results of our algorithms to evaluate the effectiveness of both query aggregation and batching. We present the performance of three algorithms for comparison. We refer to the algorithm that employs both query aggregation and batching as Aggregate+Batch. In addition, we also collect the execution time of the algorithm with query aggregation but no batching where we refer to this algorithm as Aggregate. The baseline algorithm evaluates point-to-point queries (with pruning) one by one, we refer to this algorithm as PTP-OneByOne. For all three algorithms, we present the data for the largest number of query vertices for each graph (i.e., 128 for LJ and PK, 64 for TTW, and 32 for TT).

First, let us consider the results of the algorithms for the *forward direction*. Table 13 presents the total execution time of the baseline algorithm while Table 14 gives the execution times of Aggregate and Aggregate+Batch algorithms as well as their speedups over the baseline algorithm. As we can see, both query aggregation and batching contribute to the speedup significantly. Query aggregation alone gives speedups ranging from 16.46× (SSWP on TT) to 48.64×(SSSP on PK). When batching is combined with aggregation, the speedups are further boosted to 28.81× (SSSP on TT) to 166.27× (SSWP on PK).

We also present the speedups of Aggregate and Aggregate+Batch over the baseline using algorithms running in the backward direction over the edge-reverse graph and compare the results of backward execution with that of forward execution. Table 15 shows the speedup that can be achieved by selecting the faster direction over

Table 9
Percentage of Vertex Values that become **Stable** due to Sharing Updates.

Graph	Batch Sizes	SSWP	Viterbi	BFS	Top 2 & 1 SSSP
TTW	4	99.99	99.99	28.95	6.93 - 6.93
	8	99.99	99.99	25.14	7.38 - 7.41
	16	99.99	99.99	22.07	5.21 - 5.25
TT	4	99.99	99.99	23.71	16.65 - 16.78
	8	99.99	99.99	20.57	19.85 - 20.03
	16	99.99	99.99	18.14	13.61 - 13.82
LJ	4	99.99	99.64	7.64	2.21 - 1.03
	8	99.99	99.64	6.56	2.01 - 1.20
	16	99.99	99.64	5.61	2.53 - 1.40
PK	4	99.99	99.63	6.26	0.88 - 1.92
	8	99.99	99.63	6.11	1.01 - 2.03
	16	99.99	99.63	5.38	3.24 - 4.35
Average		99.99	99.80	12.87	6.10 - 6.18

Table 10
Sensitivity to **p** Value: Cost of Phase II, Benefit of Phase II, Speedup of Sharing Over Batching Alone on 256 Queries.

p	SSWP			Viterbi		
	TTW					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.08	0.97	8.92×	0.09	0.97	8.47×
2	0.08	0.81	3.74×	0.07	0.84	4.21×
3	0.08	0.49	1.70×	0.08	0.50	1.72×
	TT					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.06	0.98	12.26×	0.06	0.98	12.64×
2	0.06	0.88	5.61×	0.05	0.89	6.39×
3	0.06	0.57	2.05×	0.06	0.59	2.17×
	LJ					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.12	0.96	6.43×	0.12	0.92	5.14×
2	0.13	0.94	5.26×	0.09	0.91	5.48×
3	0.12	0.88	4.19×	0.10	0.85	3.89×
	PK					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.09	0.98	8.76×	0.08	0.96	8.22×
2	0.09	0.95	7.25×	0.07	0.95	8.57×
3	0.09	0.83	3.91×	0.08	0.87	4.68×

Table 11
Changes in Batch Execution Time (seconds): Dynamically Selected → From Other Batch.

Graph::Alg.	Batch 2	Batch 3	Batch 4
TTW::SSWP	14.1 → 14.4	12.9 → 14.1	12.3 → 13.3
TTW::Viterbi	15.1 → 16.4	13.4 → 14.5	14.4 → 14.4
TT::SSWP	17.5 → 18.6	16.2 → 17.5	16.2 → 17.3
TT::Viterbi	18.6 → 18.5	17.5 → 17.6	17.1 → 17.3

Table 12
Number of Unique Shared Vertices Selected Over Four Batches: Min < Actual < Max.

Graph::Alg.	$\|SSET\| = 1$	$\|SSET\| = 3$	$\|SSET\| = 5$
TTW::SSWP	1 <3 <4	3 <7 <12	5 <9 <20
TTW::Viterbi	1 <3 <4	3 <7 <12	5 <9 <20
TT::SSWP	1 <2 <4	3 <8 <12	5 <9 <20
TT::Viterbi	1 <2 <4	3 <8 <12	5 <9 <20

the slower direction. As we can see, the difference between forward and backward execution is more significant for large graphs with fewer query vertices (i.e., TTW with 32 query vertices and

Table 13
Running Time (Seconds) of Baseline in Forward Direction. Number of Query Vertices: 128 for LJ and PK, 64 for TTW, 32 for TT.

Graph	SSWP	Viterbi	BFS	SSSP
TTW	2180	2306	1001	3351
TT	202	229	126	243
LJ	350	413	144	469
PK	625	613	151	567

TT with 32 query vertices) while the difference between two directions is smaller for smaller input graphs with larger number of query vertices (i.e., LJ and PK with 128 queries). A possible reason for the difference between TTW/TT and LJ/PK is the difference in their graph structure. In particular, the difference between out-degrees and indegrees of the query vertices of TTW/TT is much larger than that of LJ/PK.

4.2.3. Sensitivity to number of query vertices

Fig. 4 presents execution times for varying number of query vertices for LJ. The general trend is that speedup increases as the number of query vertices increases. This is because greater

Table 14
Running Time and Speedup of Aggregate and Aggregate+Batch over baseline PTP-OneByOne. In each cell, the Left Number is Execution Time in Seconds while the Right Number is Speedup over Baseline.

Algorithm	SSWP		Viterbi		BFS		SSSP	
	TTW							
Aggregate	106	20.62×	117	19.70×	25	40.29×	159	21.05×
Aggregate+Batch	25	86.17×	28	83.85×	17	60.58×	38	88.36×
	TT							
Aggregate	12	16.46×	13	17.68×	5	24.18×	15	16.66×
Aggregate+Batch	7	30.82×	7	31.41×	3	41.99×	8	28.81×
	LJ							
Aggregate	10	34.03×	13	32.40×	4	33.00×	16	30.16×
Aggregate+Batch	3	121.54×	3	118.12×	2	84.56×	4	118.35×
	PK							
Aggregate	14	46.32×	14	44.56×	3	45.31×	12	48.64×
Aggregate+Batch	4	166.27×	4	158.36×	1	134.17×	4	159.00×

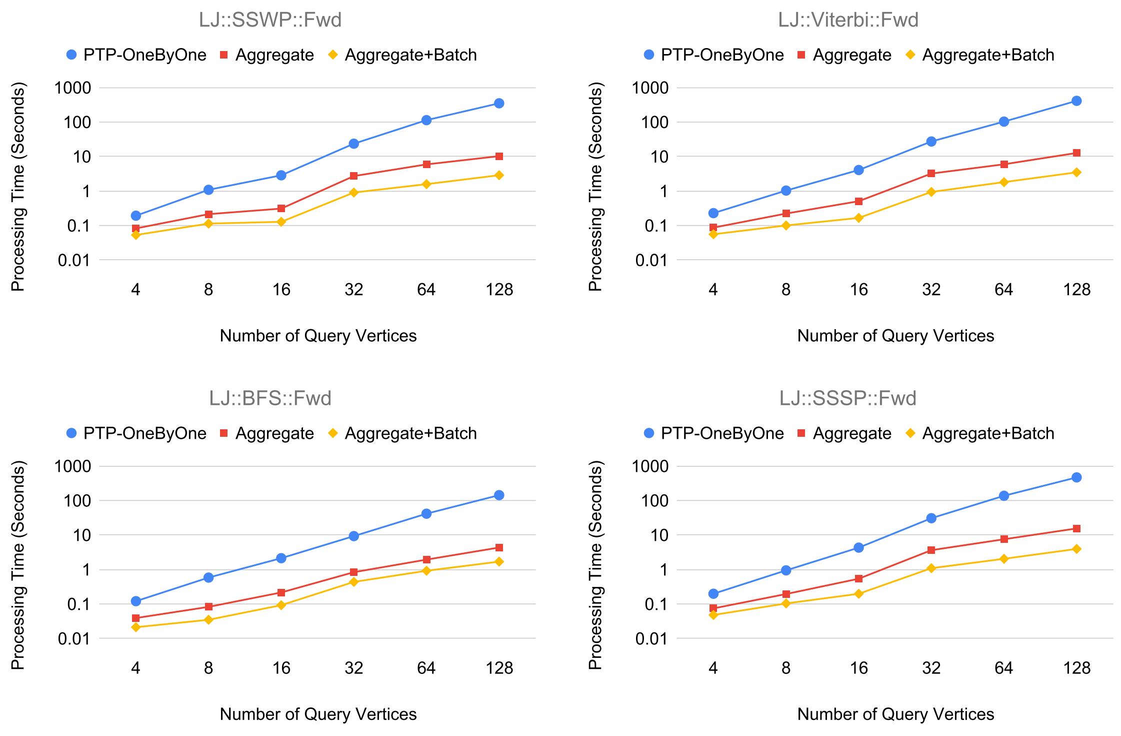


Fig. 4. Total Query Execution Times of PTP-OneByOne vs. Aggregate vs. Aggregate+Batch.

amounts of redundant computation can be eliminated via query aggregation and greater amount of runtime overhead is amortized via batching. This is reflected in Fig. 4 as the gaps between the baseline and Aggregate (for aggregation) and between Aggregate and Aggregate+Batch (for batching) increase as the number of query vertices increases. For instance, as shown in Table 16, in the case of SSWP on LJ, the speedup from aggregation and speedup from batching is 2.34× and 1.55× respectively for 4 query vertices. The corresponding speedups grow to 19.18× and 3.57× as the number of query vertices increases to 64.

4.2.4. Accuracy for direction prediction

As shown in Table 15, the running time of Aggregate+Batch can differ a lot when evaluating in different directions. Thus it is important to figure out the faster direction. Since we observe that the number of iterations can be as small as a few iterations for our workload, we decide to use static direction prediction heuristic in the beginning before the iterative computation starts.

We evaluated the following two degree-based heuristics. The underlying assumption is that lower degree indicates less work to do which has been verified in prior work PnP [29]. In the discussion below, k denotes the number of query vertices.

In the first heuristic, for each of the k query vertices we compare their outdegree and indegree. If outdegree is less than indegree, the vote for forward execution increases by one. Otherwise, the vote for backward execution increases by one. After examining every query vertex, the direction with more votes will be selected as the desired direction.

In the second heuristic, for each of the k^2 point-to-point queries we compare the outdegree of the source vertex and the indegree of the destination vertex. If the former is less than the latter, the vote for forward execution increases by one. Otherwise, the vote

Table 15
Speedup of Faster Direction over Slower Direction using Aggregate+Batch.

Algorithm	SSWP	Viterbi	BFS	SSSP
	TTW			
PTP-OneByOne	2.81×	2.75×	2.56×	3.28×
Aggregate	3.35×	3.35×	2.32×	3.61×
Aggregate+Batch	2.24×	2.32×	2.22×	2.25×
	TT			
PTP-OneByOne	3.51×	3.43×	3.38×	3.80×
Aggregate	5.20×	4.80×	3.59×	5.29×
Aggregate+Batch	2.97×	3.04×	2.68×	3.04×
	LJ			
PTP-OneByOne	1.19×	1.11×	1.11×	1.21×
Aggregate	1.27×	1.13×	1.02×	1.13×
Aggregate+Batch	1.18×	1.02×	1.01×	1.07×
	PK			
PTP-OneByOne	1.02×	1.02×	1.00×	1.08×
Aggregate	1.01×	1.04×	1.02×	1.08×
Aggregate+Batch	1.02×	1.04×	1.01×	1.05×

Table 16
Execution Times (Seconds) of SSWP on LJ for Varying # of Query Vertices.

# Query Vertices	4	8	16	32	64	128
PTP-OneByOne	0.19	1.09	2.86	23.62	113.74	350.38
Aggregate	0.08	0.21	0.31	2.72	5.93	10.29
Aggregate+Batch	0.05	0.11	0.13	0.91	1.59	2.88

Table 17
Prediction Rate - First Heuristic.

Graph	SSWP	Viterbi	BFS	SSSP
TTW	100.00%	100.00%	100.00%	100.00%
TT	100.00%	100.00%	100.00%	100.00%
LJ	66.67%	100.00%	50.00%	83.33%
PokeC	50.00%	50.00%	33.33%	50.00%

for backward execution increases by one. After examining all the point-to-point queries, the direction with more votes will be predicted as the faster direction.

For each combination of benchmark application and input graph, we compute the prediction rate based on the overall results on different numbers of query vertices. For instance, in the case of SSWP on LJ, the prediction rate is based on six data points which are collected for six different numbers of query vertices 4, 8, 16, 32, 64, and 128.

Tables 17 and 18 present the prediction rates for the two heuristics. Both prediction heuristics give very good results for TTW and TT and reasonably good results for LJ and PK. When prediction heuristics work less efficiently, the forward and backward execution times are usually close to each other (e.g., Viterbi on PokeC for which the plots of running time in different directions are shown in Fig. 5). In contrast, the difference between execution times in forward and backward is usually much more significant when the prediction rate is more accurate (e.g., SSWP on TTW, for which difference between directions is shown in Fig. 5). Therefore, with our prediction heuristics, even with misdirection, we still get good performance in the end.

5. Related works

Multi Query Frameworks. Recently, MultiLyra [15] and its extensions in BEAD [16] were developed to simultaneously evaluate a batch of iterative graph queries. There are important differences between the algorithms developed in this paper and MultiLyra/BEAD. First, MultiLyra and BEAD are frameworks for distributed systems and hence its emphasis is on amortizing *communication costs* between machines of a cluster while in this paper we show how batching can be deployed on a single multicore shared-memory machine to amortize overhead costs. Second, we show how to *dynamically* identify shared queries and exploit them to amortize computation costs of queries in a *single batch*. MultiLyra presents a limited algorithm that profiles multiple batches to find *fixed* shared queries that it uses to help speedup future batches. Thus, it cannot be used to speedup a single batch of queries and it cannot select shared queries that are customized to the batch being evaluated. Also in [24] authors show that a batch of BFS queries starting from different source vertices can be simultaneously evaluated efficiently. In [8] authors group vertices into multiple batches to reduce message passing and remote memory access in computing pruned landmark labels. However, they do not exploit sharing and are aimed at a specific application.

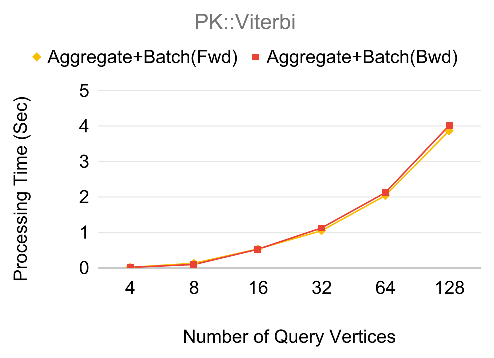
Table 18
Prediction Rate - Second Heuristic.

Graph	SSWP	Viterbi	BFS	SSSP
TTW	100.00%	100.00%	100.00%	100.00%
TT	100.00%	100.00%	100.00%	100.00%
LJ	66.67%	100.00%	50.00%	83.33%
PokeC	66.67%	33.33%	83.33%	100.00%

Congra [17] schedules a group of concurrent queries to fully utilize the memory bandwidth while preventing contention between different queries. It relies upon offline profiling with different number of threads to determine the scalability and memory bandwidth consumption of different graph algorithms on different input graphs. Multiple queries are processed by creating different processes for different queries where each process has suitable number of threads. This approach thus exploits available system resources fully. In contrast, SimGQ+ does not require offline profiling but is entirely online, lightweight, and enjoys additional benefits from sharing and batching because it does not use multiple processes. Unlike our sharing of computation across queries, Congra does not exploit shared computations across multiple queries in a batch and thus it does not reduce the amount of computation in terms of number of updates or active vertices scheduled. As for batching, we group the updates from different queries on the same vertex together to achieve better cache performance, while Congra cannot do so as execution of each query is decoupled from other queries. Other works on concurrent query processing include CGraph [31] that merely studies the opportunity to share the graph in the context of out-of-core system and Seraph [27] that studies the opportunity to share the graph and emphasizes its capability to help in fault tolerance rather than performance.

Recent works that address the problem of evaluating multiple point-to-point graph queries are Quegel [30] and PnP [29]. Quegel achieves higher throughput by simultaneous evaluating multiple queries in a pipelined fashion on a distributed system. In comparison, the batching and aggregation scheme we propose exploits much greater level of parallelism and exploits new optimizations some of which are inspired by PnP [29]. Note that PnP evaluates queries one by one. Finally, Wonderland [33] supports both general and point-to-point queries; however, it does not support sharing.

Graph Databases and Query Systems. There has been a great deal of work on graph based query languages (e.g., Gremlin [21]) and query support in graph databases (e.g., Neo4J and DEX [4,1]) that enable graph traversals and joins via lower-level graph primitives (e.g., vertices, edges, etc.). However, they are not efficient for iterative graph algorithms over large graphs. For example, although Neo4J supports shortest path queries, as shown in [30], Neo4J runs out of memory for large graphs (e.g., TT used in this paper) and although it can handle smaller graphs (e.g., LJ used in this paper) it runs extremely slowly taking tens of thousands of seconds in com-



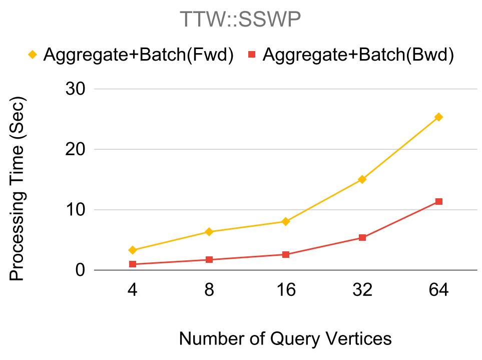


Fig. 5. Aggregate+Batch (Forward) vs. Aggregate+Batch (Backward) Running Times.

parison to just few seconds required by our system. The strength of above systems lies in their ability to program wide range of queries especially neighborhood queries [19,20]. In [34] authors present SPath, an indexing method which leverages decomposed shortest paths around neighborhood of each vertex as basic indexing unit, to accelerate subgraph matching queries. SPath performs very large amounts of precomputation (to enable the optimization) before it can begin to answer queries. In fact the overhead is substantial – comparable to solving a very large number of queries. SimGQ+ requires no precomputation, rather it identifies shared computation for a batch of queries such that performing it once leads to net reduction in execution time.

6. Conclusions

We developed techniques for simultaneous evaluation of large batches of iterative point-to-all and point-to-point graph queries. By employing batching, the overhead costs of query evaluation are amortized across the queries. By employing sharing for point-to-all queries and query aggregation for point-to-point queries, the cost of computations involving shared queries are amortized across the original batch of queries. Our experiments show that batching, sharing, and aggregation contribute to the substantial speedups.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Rajiv Gupta reports financial support was provided by National Science Foundation.

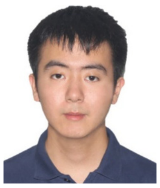
Acknowledgments

This work is supported by NSF grants CCF-2002554, CCF-1813173, and CCF-2028714 to the Univ. of California Riverside.

References

[1] A.B. Ammar, Query optimization techniques in graph databases, Int. J. Database Manag. Syst. 8 (4) (August 2016).
[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, X. Lan, Group formation in large social networks: membership, growth, and evolution, KDD (2006) 44–54.
[3] M. Cha, H. Haddadi, F. Benevenuto, P.K. Gummadi, Measuring user influence in twitter: the million follower fallacy, Intl. AAAI Conf. Web Soc. Media 10 (10–17) (2010) 30.
[4] Developers, Neo4J, Graph NoSQL Database, 2012.
[5] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in: OSDI'12.
[6] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, R. Zadeh Wtf, The who to follow service at twitter, in: WWW, 2013, pp. 505–514.
[7] G. He, H. Feng, C. Li, H. Chen, Parallel simrank computation on large graphs with iterative aggregation, in: KDD, 2010, pp. 543–552.
[8] R. Jin, Z. Peng, W. Wu, F. Dragan, G. Agrawal, B. Ren, Parallelizing pruned landmark labeling: dealing with dependencies in graph algorithms, ACM ICS 11 (2020) 1–13.
[9] H. Kwak, C. Lee, H. Park, S. Moon, What is Twitter, a social network or a news media?, in: WWW, 2010, pp. 591–600.
[10] A. Kyrola, G. Blelloch, C. Guestrin, GraphChi: large-scale graph computation on just a PC, in: USENIX OSDI, pp. 31–46, 2012.
[11] J. Lember, D. Gasbarra, A. Koloydenko, K. Kuljus, Estimation of Viterbi path in Bayesian hidden Markov models, arXiv:1802.01630, Feb. 2018, pp. 1–27.
[12] J. Leskovec, Stanford large network dataset collection, http://snap.stanford.edu/data/index.html, 2011.
[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed GraphLab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (2012) 8.
[14] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: ACM SIGMOD Int. Conf. on Management of Data, 2010.
[15] A. Mazloumi, X. Jiang, R. Gupta MultiLyra, Scalable distributed evaluation of batches of iterative graph queries, in: IEEE International Conference on Big Data, Dec. 2019, pp. 349–358.
[16] A. Mazloumi, C. Xu, Z. Zhao, R. Gupta BEAD, Batched evaluation of iterative GraphQueries with evolving analytics demands, in: IEEE International Conference on Big Data, Dec. 2020.
[17] P. Pan, C.Li. Congra, Towards efficient processing of concurrent graph queries on shared-memory machines, in: IEEE ICCD, 2017.
[18] D. Nguyen, A. Lenharth, K. Pingali, A lightweight infrastructure for graph analytics, in: ACM SOSP, 2013, pp. 456–471.
[19] M. Potamias, F. Bonchi, A. Gionis, G. Kollios, k-nearest neighbors in uncertain graphs, Proc. VLDB Endow. (2010).
[20] A. Quamar, A. Deshpande, J. Lin, NScale: neighborhood-centric analytics on large graphs, VLDB J. 7 (13) (2014) 1673–1676.
[21] M.A. Rodriguez, The gremlin graph traversal machine and language (invited talk), in: Symp. on Database Prog. Languages, 2015, pp. 1–10.
[22] J. Shun, G. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in: ACM PPoPP, 2013, pp. 135–146.
[23] L. Takac, M. Zabovsky, Data analysis in public social networks, in: International Scientific Conference and International Workshop Present Day Trends of Innovations, 2012, pp. 1–6.
[24] M. Then, M. Kaufmann, F. Chirigati, T-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, H.T. Vo, The more the merrier: efficient multi-source graph traversal, Proc. VLDB Endow. (2015).
[25] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[26] K. Vora, S-C. Koduru, R. Gupta ASPIRE, Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM, in: SIGPLAN OOPSLA, October 2014, pp. 861–878.
[27] J. Xue, Z. Yang, Z. Qu, S. Hou, Y. Dai, Seraph: an efficient, low-cost system for concurrent graph processing, in: HPDC, 2014.
[28] C. Xu, A. Mazloumi, X. Jiang, R. Gupta SimGQ, Simultaneously evaluating iterative graph queries, in: IEEE HiPC, 2020.
[29] C. Xu, K. Vora, R. Gupta PnP, Pruning and prediction for point-to-point iterative graph analytics, in: ACM ASPLOS, 2019.
[30] D. Yan, J. Cheng, M.T. Ozsu, F. Yang, Y. Lu, J.C.S. Lui, Q. Zheng, W. Ng, A general-purpose query-centric framework for querying big graphs, Proc. VLDB Endow. 9 (7) (2016) 564–575.
[31] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, H. Liu, Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing, in: USENIX ATC, 2018.

[32] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, S. Amarasinghe, GraphIt: a high-performance graph DSL, in: PACM 2, OOPSLA, 2018.
[33] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, K. Chen, Wonderland: a novel abstraction-based out-of-core graph processing system, in: ACM ASPLOS, 2018, pp. 608–621.
[34] P. Zhao, J. Han, On graph query optimization in large networks, Proc. VLDB Endow. 3 (1–2) (2010) 340–351.



Chengshuo Xu received a PhD degree in computer science from University of California, Riverside, Riverside, USA, in 2021. Prior to that, he received a BS degree in computer science from the University of Macau, Macao SAR, in 2016. His research interests include high-performance computing, graph analytics, and AI accelerators.

Abbas Mazloumi received the BS degree in computer engineering from the University of Mazandaran, Babolsar, Iran, in 2009, and the MS degree in computer architecture from the University of Tehran, Tehran, Iran, in 2014. He is currently working toward the Ph.D. degree in the Department of Computer Science and Engineering at the University of California, Riverside, California, USA. His research interests include computer architecture, networks-on-chip, high-performance computing, graph analytics, and graph AI.

Xiaolin Jiang received a bachelor's degree in computer science and technology from Shandong University, Jinan, China. She is currently working toward the PhD degree with the CSE Department at the University of California, Riverside. Her research interests include graph processing, and are broadly in the area of parallel computing, and systems.

Rajiv is a Distinguished Professor and the Amrik Singh Poonian Chair in Computer Science at the University of California, Riverside. His research interests include Compilers, Architectures, and Runtimes for Parallel Systems. He has supervised PhD dissertations of 35 students including two winners of the ACM SIGPLAN Outstanding Doctoral Dissertation Award. Papers coauthored by Rajiv with his students have been selected for: inclusion in 20 Years of PLDI (1979-1999), a best paper award in PACT 2010, and a distinguished paper award in ICSE 2003. Rajiv is a Fellow of the IEEE (2008), ACM (2009), and AAAS (2011). He received the NSF Presidential Young Investigator Award (1991) and UCR Doctoral Dissertation Advisor/Mentor Award (2012). He has chaired several major conferences including FCRC, PLDI, HPCA, ASPLOS, PPoPP, CGO, CC, HiPEAC, and LCTES. Rajiv served as a member of a technical advisory group on networking and information technology created by US President's Council of Advisors on Science and Technology (PCAST) (2006-2007).