PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics

Chengshuo Xu
Univ. of California, Riverside
cxu009@ucr.edu

Keval Vora
Simon Fraser University
keval@cs.sfu.edu

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

Abstract

Frequently used parallel iterative graph analytics algorithms are computationally expensive. However, researchers have observed that applications often require *point-to-point* versions of these analytics algorithms that are less demanding. In this paper we introduce the **PnP** parallel framework for iterative graph analytics that processes a stream of point-to-point queries with each involving a single source and destination vertex pair. The efficiency of our framework is derived from the following two novel features: **online Pruning** of graph exploration that eliminates propagation from vertices that are determined to not contribute to a query's final solution; and **dynamic direction Prediction** for solving the query in either forward (from source) or backward (from destination) direction as their costs can differ greatly. **PnP** employs a two-phase algorithm where, Phase 1 briefly traverses the graph in both directions to predict the faster direction and enable pruning; then Phase 2 completes query evaluation by running the algorithm for the chosen direction till it converges. Our experiments show that **PnP** responds to queries rapidly because of accurate direction selection and effective pruning that often offsets the runtime overhead of direction prediction. **PnP** substantially outperforms Quegel, the only other point-to-point query evaluation framework. Our experiments on multiple benchmarks and graphs show that **PnP** on a single machine is 8.2× to 3116× faster than Quegel on a cluster of four machines.

CCS Concepts • **Information systems** → **Social networks**; • **Computing methodologies** → **Shared memory algorithms**.

Keywords point-to-point graph queries; computation pruning; direction prediction

ASPLOS'19, April 13–17, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6240-5/19/04...$15.00
https://doi.org/10.1145/3297858.3304012

ACM Reference Format:
Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304012

1 Introduction

Parallel iterative frameworks are used to compute important properties for large real-world graphs. Such frameworks have been developed for both shared-memory platforms (Ligra [31], Galois [26], GRACE [39], GridGraph [45], X-Stream [30] etc.) and distributed clusters (PowerGraph [10], KickStarter [36], CoRAL [35] etc.). Even though iterative graph analytics algorithms are highly parallel, for large graphs they are expensive due to their *exhaustive* nature (e.g., shortest path algorithm starts from a single source and computes shortest paths to *all* destination vertices).

Recently Yan et al. [41] 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 point-to-point queries over pairs of certain important shoppers. Thus, given a pair of distinct vertices (s, d) in a graph, we are interested in computing point-to-point versions of standard computations such as, shortest path from s to d, widest path from s to d and number of paths from s to d. Yan et al. developed the Quegel [41] framework to solve point-to-point queries.

Although Quegel presents a solution for evaluating point-to-point queries, it is far from optimized. First, Quegel does a significant level of wasteful work as it does not prune traditional *one source to all destinations* computation to achieve point-to-point subcomputation. Second, it does not recognize that evaluation times of point-to-point queries in backward and forward directions can greatly differ. In contrast we present **PnP** that addresses the above drawbacks and delivers significant speedups over Quegel.

Quegel supports Hub^2 [14] precomputation to speedup evaluation of individual queries. However, this approach has multiple drawbacks that limits its utility. The experimental data reported in [41] shows that Hub^2 precomputation is *expensive*. Moreover, in the common scenario where graph structure mutates, the Hub^2 precomputation must be repeated making Quegel unsuitable for streaming (changing) graphs. While KickStarter's value-dependence based trimming strategies [36] can be used to accelerate Hub^2 computation, the repetitive trimming of Hub^2 information does not justify separating it out as a preprocessing step for relatively-inexpensive queries. Finally the Hub^2 [14] precomputation is specifically designed for accelerating *shortest path* queries on graphs where all *edge weights are the same.* This limits its use both in terms of types of *queries* and *graphs.*

In this paper we present **PnP** framework that avoids all the limitations of Quegel and efficiently computes point-to-point versions of wide range of queries on weighted and unweighted graphs. **PnP** does not require any precomputation thus allowing graph changes in between queries. To quickly respond to queries **PnP** uses dynamic techniques for optimizing query evaluation. In particular, it uses two *general* dynamic techniques: ***online Pruning*** of graph exploration that eliminates propagation from vertices determined to not contribute to a query's final solution; and ***dynamic direction Prediction*** method for choosing between solving the query in forward (from source) or backward (from destination) direction as their costs can differ significantly based on the graph structure and computation behavior.

We carry out an experimental study (§2) that shows how query characteristics and the direction of evaluation impact runtime. Guided by the observations, we propose **PnP**'s *two-phase* algorithm (§3) that delivers fast evaluation times across queries with differing characteristics. Phase 1 briefly traverses the graph in both forward and backward directions originating from source and destination vertices. By monitoring progress in both directions during this phase we are able to predict the faster direction highly accurately and compute information that enables pruning. Phase 2 completes the point-to-point computation by running the algorithm, with pruning enabled, in the chosen direction to convergence.

There is prior work on graph based query languages (e.g., Gremlin [29]) and query support in graph databases (e.g., Neo4J and DEX [2, 9, 20]) that enable graph traversals and joins via lower-level graph primitives (e.g., vertices, edges, etc.). The strength of these systems is their versatility. They are widely used for solving neighborhood queries and looking for patterns in graphs [22, 27, 28, 38, 40, 42]. However, the generality comes at a cost – they are not optimized for performance for iterative graph analytics whose scope extends across the entire graph. For example, although Neo4J supports point-to-point shortest path queries, as shown in [41], Neo4J runs out of memory for large graphs (e.g., Twitter-TT [5] used in this paper) and although it can handle small graphs (e.g., LiveJournal-LJ [3] used in this paper) it runs extremely slowly taking tens of thousands of seconds in comparison to just few seconds required by **PnP**.

The key contributions of this paper are as follows:

(**sPr**) We introduce *simple Pruning* based point-to-point query evaluation in either direction. We study the impact of query characteristics and evaluation direction on execution time for multiple algorithms on graphs. (§2)

(**2Ph**) Guided by observations from the study of sPr algorithm, we develop a *two-phase* algorithm that in Phase 1 quickly identifies preferable direction and basis for pruning and then in Phase 2 evaluates the query in the *predicted direction* with greatly *enhanced pruning.* (§3)

(**PnP** framework) that is easy to use. It requires user to simply provide two functions for the computation and three for pruning. It has been implemented as an extension of the Ligra [31] framework. (§2 & §3)

(**Evaluation**) **PnP** responds to expensive queries rapidly (few seconds) because direction selection is accurate and pruning is effective. When direction is mispredicted, it is for queries where direction has relatively small impact on performance. Alsi pruning often offsets the overhead of Phase 1. Finally, **PnP** greatly outperforms Quegel. (§4)

2 Simple Pruning (sPr) Based Study of Point-to-Point Query Characteristics

In this section we present an algorithm for computing point-to-point queries with *simple pruning* (sPr) and then analyze the runtime characteristics of the algorithm on 10,000 queries each for four input graphs and multiple analytics problems. This large scale study allows us to uncover runtime characteristics that enable us to develop a new two-phase algorithm that dynamically predicts and adapts execution to deliver highly optimized performance across all types of queries. Note that prior work has been limited in its scope – Quegel uses 1000 shortest path queries [41]; thus, the observations exploited in this work eluded prior work on Quegel.

Each point-to-point query is of the form $Q(s \leadsto d, G)$ where G is a directed graph, s is the chosen source vertex, and d is the chosen destination vertex. Thus, we compute the desired property Q with respect to $s \leadsto d$ (e.g., Shortest Path from s to d, Widest Path from s to d etc.). To avoid negative-weight cycles, edge weights are assumed to be positive. In comparison to standard iterative algorithms, the iterative algorithm for point-to-point query has two distinct features: it employs *pruning* and it provides *direction choice.*

The ***online pruning*** of graph exploration is enabled by the observation that point-to-point evaluation algorithm only needs to achieve convergence for $s \leadsto d$ as opposed to all possible (destination) vertices. Pruning dynamically eliminates wasteful computation and propagation that is determined not to contribute to the final solution for the query. Pruning leads to early termination relative to the standard iterative algorithm. The pruning strategy is easily identifiable for *monotonic* problems, i.e. the solution for the property value being computed monotonically increases or decreases through the iterations of the algorithm before stabilizing to its final value.

In evaluating the query we have ***direction choice***. That is, we can either compute $Q(s \leadsto d, G)$ in forward direction (i.e.,

Algorithm 1 Point-to-Point with Simple Pruning (**sPr**).

```
function EVALUATE ( Q ( $s \rightsquigarrow d$, $Graph$ ) )
    ▷ Initialize active set of vertices
    ACTIVE ← INITIALIZE ( Q ( $s \rightsquigarrow d$ ) )
    ▷ Iterate
    while ACTIVE $\neq \phi$ do
        ACTIVE ← PROCESS ( ACTIVE, $d$, $Graph$ )
    end while
end function

function PROCESS ( ACTIVE , DEST, $Graph$ )
    NEWACTIVE ← $\phi$
    ▷ Compute new property values
    for all $v \in$ ACTIVE do
        for all $e \in Graph.outEdges(v)$ do
            $changed$ ← EDGEFUNCTION ($e$)
            if $changed$ and DONOTPRUNE ($e.dest$, DEST) then
                NEWACTIVE ← NEWACTIVE ∪ {$e.dest$}
            end if
        end for all
    end for all
    return NEWACTIVE
end function
```

starting from s and propagating forward along the directed edges in G), or alternatively, we can compute the query in backward direction as $Q(d \rightsquigarrow s, \widehat{G})$ where we start at d and propagate forward in $\widehat{G}$, the edge reversed graph corresponding to G (i.e., $\widehat{G}$ is obtained by reversing the direction of all edges in G). We show that direction impacts execution time.

It is crucial to note that point-to-point queries can also be formulated on undirected graphs. While the techniques presented in this paper work for undirected graphs as well, we present them using directed graphs for simpler exposition. In particular, our direction monitoring and selection techniques are primarily based on the direction of value propagation from/to source/destination vertices, and hence, remains oblivious to the graph being directed or undirected.

In Algorithm 1 EVALUATE carries out *push-style* evaluation of a query for vertex pair ($s \rightsquigarrow d$) starting at the source vertex s by iteratively processing active vertices by calling PROCESS till the set of active vertices becomes empty and propagation ceases. In contrast to standard algorithm, it constructs a *pruned active set*. Pruning is achieved by comparing the newly computed value of each vertex v with that of destination vertex d (line 16). If it is determined that propagating v's current value through the graph cannot cause a change in d's value, then propagation of v's value is pruned. Consider the evaluation of shortest path from s to d. At any execution point, d's current value represents the length of the shortest path from s to d that has been found so far. If v's value, that represents the length of the shortest path from s to v, is greater than or equal to d's value then it need not be propagated as it can only discover longer paths to d.

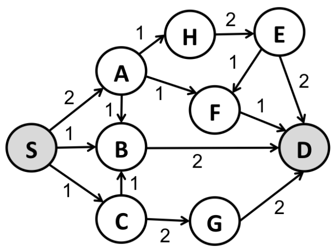

Figure 1. Shortest path evaluation $SP(S \rightsquigarrow D)$.

ACTIVE	A	B	C	D	E	F	G	H
–	∞	∞	∞	∞	∞	∞	∞	∞
S (= 0)	2	1	1	∞	∞	∞	∞	∞
A,B,C	2	1	1	**3**	∞	3	3	3
	Early Termination							
D,F,G,H	2	1	1	**3**	5	3	3	3
E	2	1	1	**3**	5	3	3	3
ϕ	*Normal Termination*							

Table 1. Shortest Path Query: Forward sPr.

ACTIVE	S	A	B	C	E	F	G	H
–	∞	∞	∞	∞	∞	∞	∞	∞
D (=0)	∞	∞	2	∞	2	1	2	∞
B,E,F,G	**3**	2	2	3	2	1	2	4
A	**3**	2	2	3	2	1	2	4
ϕ	*Early Termination*							

Table 2. Shortest Path Query: Backward sPr.

The above framework relies upon the user to provide two essential functions for each algorithm: **EDGEFUNCTION** is the main computation function that updates the property value of a destination vertex and returns whether the update succeeded or not (CASMIN(a, b) sets $a = b$ if $b < a$ atomically using compare-and-swap); and **DONOTPRUNE** which determines whether the propagation can be pruned. For illustration, the two functions for the shortest path point-to-point query SP($s \rightsquigarrow d, G$) are given below.

EDGEFUNCTION (e): CASMIN($e.dest.value$, $e.source.value + e.weight$)

DONOTPRUNE (v, d): $v.value < d.value$

To solve the query in backward direction we can instead compute SP($d \rightsquigarrow s, \widehat{G}$).

The example in Figure 1 illustrates the above algorithm and the early termination it achieves via pruning. Table 1 shows the progress of the *shortest path* computation from S to D, iteration by iteration. In each row the set of active vertices that are processed is presented along with the updated values following their processing. The values marked in green are those that have changed requiring further propagation while at the same time are not pruned; thus they are used to compute the ACTIVE set for the following iteration. The values marked in red are those that have changed but pruned because they are greater than or equal to the value of the D, the destination vertex. Therefore pruning of vertices F, G, H in row three leads to early termination. If pruning is

Graphs	#Edges	#Vertices	#Queries
Twitter (TTW) [16]	1.5B	41.7M	10K
LiveJournal (LJ) [3]	69M	4.8M	10K
Twitter (TT) [5]	2.0B	52.6M	10K
PokeC (PK) [32]	31M	1.6M	10K

Table 3. Real world input graphs.

G	Queries	WP	SP	NP	BFS
TTW	FwdR	49.26%	44.55%	15.24%	31.07%
	BwdR	13.30%	18.01%	47.32%	31.49%
	FwdNR	20.41%	20.41%	20.41%	20.41%
	BwdNR	17.03%	17.03%	17.03%	17.03%
LJ	FwdR	10.81%	13.23%	12.84%	7.89%
	BwdR	37.41%	34.99%	35.38%	40.33%
	FwdNR	24.75%	24.75%	24.75%	24.75%
	BwdNR	27.03%	27.03%	27.03%	27.03%
TT	FwdR	41.86%	10.61%	28.86%	29.23%
	BwdR	12.40%	43.65%	25.40%	25.03%
	FwdNR	35.02%	35.02%	35.02%	35.02%
	BwdNR	10.72%	10.72%	10.72%	10.72%
PK	FwdR	3.60%	5.27%	1.65%	6.69%
	BwdR	17.30%	15.63%	19.25%	14.21%
	FwdNR	38.55%	38.55%	38.55%	38.55%
	BwdNR	40.55%	40.55%	40.55%	40.55%

Table 4. Characteristics of 10,000 queries used in experiments: Fwd – Forward faster; Bwd – Backward faster; R – Reachable; and NR – Non-reachable.

not performed the algorithm takes two additional iterations to terminate. Note that during these iterations the value for vertex D does not change further confirming that the processing of vertices that were pruned does not contribute to the query solution. Table 2 illustrates backward evaluation of the shortest path from S to D . When we compare the results of Table 2 with that of Table 1 we observe that cost of the two algorithms vary. In this case we find that the forward algorithm processes fewer active vertices (and edges) and takes fewer iterations.

Next we present results of our study. We first describe the experimental setup below.

Experimental setup – For this study we implemented our framework using Ligra [31] which uses the Bulk Synchronous Model [33] and provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. The study is based upon four algorithms – Shortest Path (SP), Widest Path (WP), Number of Paths (NP), and Breadth First Search (BFS). We use four input graphs listed in Table 3 – two are billion edge graphs (TTW, TT) and two have tens of millions of edges (LJ, PK). For each input graph, we generated 10,000 queries and used them to evaluate all algorithms. No vertex appears more than once, either as a source or destination, in these queries. Moreover, the vertices chosen as sources and destinations are selected by sampling all the vertices ordered by their degrees. All experiments were performed on a 64 core (8 sockets, each with 8 cores) machine with AMD Opteron 2.3 GHz processor 6376, 512 GB memory, and running CentOS Linux release 7.4.1708.

In this study, the 10,000 queries used are classified into four distinct categories based upon combination of two properties: (Fwd vs. Bwd) queries for which forward evaluation is faster belong to Fwd and those for which backward evaluation is faster belong to Bwd; (NR vs. R) queries that reveal that destination is non-reachable from the source belong to NR and queries where destination is reachable from source belong to R. Therefore, the queries on a given workload can be divided into four categories: FwdNR, BwdNR, FwdR, and BwdR. The distribution of the 10,000 queries based upon faster/slower direction and reachability/non-reachability is shown in Table 4. We observe there are a good number of queries of all four types. Note that numbers for NR queries are same for different benchmarks as they are mainly influenced by graph structure.

Analysis of execution times – We ran all 10,000 queries for each input on sPr versions of all four graph algorithms and collected their forward and backward evaluation times. For reachable queries sPr carries out pruning once it finds the first approximation of query solution while for non-reachable queries pruning never takes place as query has no result. Average execution times of all queries by category are given in Tables 5 (Non-Reachable) and 6 (Reachable). Figure 2 shows a representative scatter plot of the execution times (all plots are shown in later section) – the times of queries in order of FwdNR FwdR, BwdR, and BwdNR from left to write are plotted. Based upon the data we make two key observations.

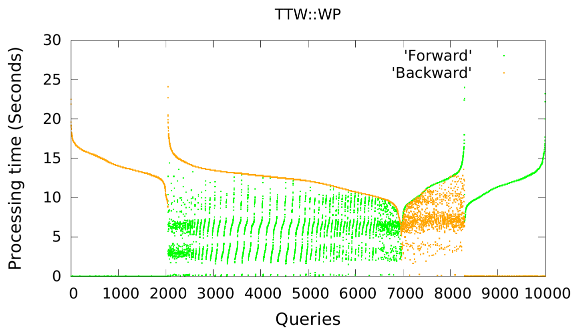

Figure 2. Forward and Backward Evaluation Times.

Observation 1 – Fwd vs. Bwd: *direction is important.* Picking the right direction for solving a query is important. From Figure 2 we can easily see that for *non-reachable* queries the difference in forward and backward execution times is consistently high and the time in the faster direction is very small; and for *reachable* queries the difference between forward and backward evaluation times varies from very large to very small. This observation holds across all algorithms and all input graphs as shown by the average times in the *faster direction* in Tables 5 and 6. Each table also presents

Graph	Queries	WP		SP		NP		BFS	
TTW	FwdNR	0.0130s	1096.52 ×	0.0129s	1137.57 ×	0.0268s	128.17 ×	0.0058s	3.42 ×
	BwdNR	0.0365s	318.25 ×	0.0258s	562.79 ×	0.0303s	145.08 ×	0.0089s	105.18 ×
LJ	FwdNR	0.0009s	484.40 ×	0.0010s	875.85 ×	0.0055s	40.90 ×	0.00096s	79.90 ×
	BwdNR	0.0009s	666.86 ×	0.0013s	620.39 ×	0.0071s	34.58 ×	0.00123s	67.12 ×
TT	FwdNR	0.0191s	772.85 ×	0.1771s	88.69 ×	0.0551s	124.26 ×	0.0170s	65.68 ×
	BwdNR	0.0282s	620.53 ×	0.0154s	1560.24 ×	0.0595s	117.32 ×	0.0299s	44.69 ×
PK	FwdNR	0.0005s	250.34 ×	0.0004s	458.73 ×	0.0024s	72.98 ×	0.0004s	76.02 ×
	BwdNR	0.0004s	478.72 ×	0.0006s	263.00 ×	0.0023s	73.24 ×	0.0004s	75.67 ×

Table 5. (sPr on NR queries) Avg. Execution Times in Faster Direction (seconds); and Avg. Slowdown Factor in Slower Direction.

Graph	Queries	WP		SP		NP		BFS	
TTW	FwdR	5.5598s	2.21 ×	9.5778s	1.33 ×	2.2778s	1.21 ×	0.2546s	2.12 ×
	BwdR	7.5349s	1.51 ×	11.177s	1.16 ×	2.4258s	1.37 ×	0.4611s	1.43 ×
LJ	FwdR	0.2480s	2.36 ×	0.7036s	1.18 ×	0.1316s	1.16 ×	0.0437s	1.20 ×
	BwdR	0.1645s	4.90 ×	0.5869s	1.38 ×	0.1205s	1.32 ×	0.0355s	1.60 ×
TT	FwdR	7.2006s	1.94 ×	11.8350s	1.27 ×	3.8697s	1.30 ×	0.4501s	1.46 ×
	BwdR	9.5975s	1.74 ×	14.7070s	1.36 ×	4.2492s	1.27 ×	0.6047s	1.29 ×
PK	FwdR	0.0742s	1.87 ×	0.1319s	1.16 ×	0.0683s	1.10 ×	0.0175s	1.26 ×
	BwdR	0.0481s	3.68 ×	0.1125s	1.31 ×	0.0607s	1.31 ×	0.0125s	1.47 ×

Table 6. (sPr on R queries) Avg. Execution Times in Faster Direction (seconds); and Avg. Slowdown Factor in Slower Direction.

the factor by which the average execution time increases if a query is solved in the slower direction as opposed to faster direction. From Table 6 for NR queries not only is the execution in faster direction very small (tens of milliseconds), in the slower direction it is orders of magnitude slower (around a second). From Table 6 for reachable queries the average execution time in faster direction is higher (several seconds for large graphs) and the slowdown factor is lower.

Observation 2 – NR vs. R: *reachability is important.* Picking the right direction alone is not enough to achieve the best performance. We need a strategy for handling both non-reachable and reachable queries effectively. In particular, we note that FwdNR/BwdNR queries can be evaluated significantly faster than FwdR/BwdR queries – well over two and often over three orders of magnitude faster. For example, for SP on TTW, average times for FwdNR/BwdNR are 0.0129s/0.0258s while for FwdR/BwdR they are 9.5778s/11.177s. In other words, since at the start of a query evaluation we do not know whether it is NR or R, we need to design a strategy that quickly classifies it as NR or R and then appropriately handles them to get fast execution times.

Next we develop a two-phase algorithm that exploits the above observations in delivering fast evaluation of all four types of queries.

3 PnP Two-Phase Framework

The goal of this section is to develop a general algorithm that delivers execution times that are close to the execution times in the faster direction for all types of queries. Based upon the observations in the preceding section, we can set the requirements that must be met by the point-to-point query evaluation algorithm as follows:

- *RQ1: effectively handle both non-reachable and reachable queries (follows from first observation);*
- *RQ2: identify the faster direction and use it for query evaluation (follows from second observation); and*
- *RQ3: maximize the use of pruning for reachable queries for quickly responding to each query.*

In this work we develop an algorithm that by design meets *RQ1*, predicts direction to meet *RQ2*, and embodies a significantly enhanced pruning strategy to meet *RQ3*.

In general, both reachability (RQ1) and direction (RQ2) requirements must be handled dynamically as queries constructed from sampling of vertices were found to fall in all four categories (see Table 4). Clearly reachability is function of the graph structure and thus without exploring the graph at runtime we cannot determine whether a query is a NR query or R query. The choice of direction matters because the cost of forward evaluation is high if forward propagation encounters many high outdegree vertices while backward evaluation is high if backward propagation encounters many high indegree vertices. We cannot simply statically look at the graph and select the same preferred direction for all queries as the overall characteristics of G and $\widehat{G}$ are similar. In Figure 3 we plot the in-degree and out-degree distributions for the LiveJournal graph. As we can see, both in-degrees and out-degrees have similar power-law distributions. Thus, for a given query, without dynamically exploring the graph in both directions we cannot establish a basis for selecting the preferred direction. Finally for meeting requirement RQ3 we need to quickly find the first approximation of the query result as soon as possible so that pruning is enabled early and greater fraction of execution is optimized via pruning.

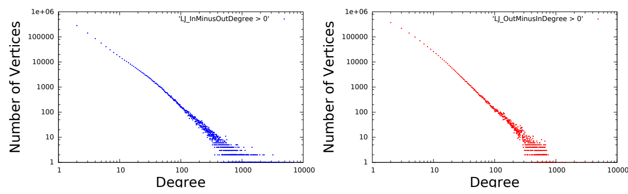

Figure 3. In- and Out-Degree Distribution of LiveJournal.

Therefore, to simultaneously meet all three requirements, we propose a two-phase algorithm such that Phase 1 dynamically and very quickly finds a suitable configuration (for NR vs. R; choose Fwd vs. Bwd) for evaluating a given query and then execution transitions to Phase 2 that evaluates the query under the selected configuration. More specifically, Phase 1 classifies the query as non-reachable or reachable (RQ1), selects preferable direction for query evaluation as forward or backward (RQ2), and enables pruning by finding a safe estimate of query's result value (RQ3). That is, Phase 1 sets the stage for requirements stated above to be met. Upon completion of Phase 1 execution moves to Phase 2 that solves the query in the preferable direction using the safe estimate of query's result to prune graph exploration. Next we discuss the design of phases in greater detail.

Phase 1: Bidirectional Exploration for Identifying Configuration. At the start we are faced with two questions: NR vs. R? and Fwd vs Bwd?. Thus, Phase 1 must decide which one to target first. Recall that in the previous section we observed that for NR queries typically one direction solves the query very quickly than any other case, i.e. for NR query in opposite direction or R query in any direction. The reason for this behavior is that typically, NR query evaluation in faster direction examines only a small fraction of the graph that is examined for its evaluation in slower direction, or evaluation of a R query examines a very large fraction of the graph. For example, for a sample of 10 NR and 10 R queries for WP on LiveJournal, we found that average percentage of vertices visited in Fwd (Bwd) direction was < 1% (87%) for NR queries and 84% (84.4%) R queries.

Given the above observation, to answer NR queries quickly we design Phase 1 to first distinguish between NR and R queries by attempting to *identify a directed path* from the source vertex and the destination vertex. Since we do not know which direction, forward or backward, is preferable, Phase 1 uses *bidirectional* exploration in both directions: forward from the source vertex; and backward from the destination vertex. During bidirectional exploration, the exploration in the fast direction quickly establishes that the query is of NR kind while relatively little time is expended in exploring the graph in the slower direction. In other words, NR queries will be identified and answered very quickly without even having to predict the preferable direction.

On the other hand, if the query is a R query, the bidirectional exploration runs a bit longer till Phase 1 determines the existence of a path from source to destination. This happens when bidirectional exploration causes some vertex to be visited from both forward and backward directions. As soon as this occurs, we know that we have a R query. The extra time spent executing allows us to observe the progress in both directions and make a choice of direction. Moreover, since a path has been fully traversed we can generate our first estimate of query's result that can be used for pruning. Now the execution transitions to Phase 2 by continuing propagation in the chosen direction while terminating propagation in the other direction, with pruning turned on right from start of Phase 2. Phase 1 is fast, pruning is maximized.

The above approach meets all the requirements as follows: (RQ1) it optimizes evaluation of both NR and R queries; (RQ2) it addresses direction problem by avoiding it for NR queries that can be quickly solved using bidirectional exploration and by predicting the preferable direction for R queries; and (RQ3) it guarantees that pruning is turned on for Phase 2.

Next we explain the details of how the desirable direction is predicted and safe approximation of query solution is computed to enable pruning at the start of Phase 2.

Direction prediction – To predict the faster direction we considered a number of measures: (Work remaining) as estimated by number of active vertices in each direction; (Work performed) as estimated by tracking the number of vertices processed in each direction; and (Hybrid) method that uses a combination of preceding two measures giving more importance to the first measure. Our experience showed that the first measure provides the highest prediction rate and thus, the direction for which there are fewer active vertices is predicted as the faster direction and used in Phase 2. An advantage of this measure is that it does not incur extra tracking overhead involved in the other measures.

Safe approximation of query solution – Since for some vertex v we have at least found a path from source to v in the forward direction and a path from v to destination in the backward direction, we can compute an estimate for query's result. When multiple vertices are visited from both directions we select the best approximation provided across all these vertices. To make use of the two-phase algorithm of our **PnP** framework, the user must provide two additional functions: one for the estimation of query result, **ESTIMATEAPPROX**, from a single vertex; and another for safe approximation for a query, **SAFEAPPROX**, from **ESTIMATEAPPROX** values of all vertices that are visited in both directions. We illustrate these by providing the functions for the shortest path query SSSP($s \rightsquigarrow d$).

ESTIMATEAPPROX(v) $\quad v.\overrightarrow{value}(s) + v.\overleftarrow{value}(d)$
SAFEAPPROX $\quad \forall\ v\ \min(\text{ESTIMATEAPPROX}(v))$

Phase 2: Query Evaluation. Upon termination of Phase 1, the execution transitions into Phase 2 where the propagation in the predicted direction is run to completion while the

Algorithm 2 Two-Phase PnP Evaluation (**2Phase**).

```
 1: function 2Phase( Query ( $s \rightsquigarrow d, G$ ) )
 2:     ▷ Initialization
 3:     VisitF (*) ← VisitB (*) ← False
 4:     FActive ← Initialize ( Query ( $s \rightsquigarrow d, G$ ) )
 5:     BActive ← Initialize ( Query ( $d \rightsquigarrow s, \widehat{G}$ ) )
 6:     safeApprox ← Query.Initialize
 7:     ▷ Phase 1
 8:     while true do
 9:         ▷ Process active vertices
10:         Processed ← FActive ∪ BActive
11:         FActive ← Process ( FActive, $d$, $G$ )
12:         BActive ← Process ( BActive, $s$, $\widehat{G}$ )
13:         ▷ Update Visit Flags of processed vertices
14:         VisitF ($v$) ← True, $\forall\ v \in$ FActive
15:         VisitB ($v$) ← True, $\forall\ v \in$ BActive
16:         ▷ Case I: Non-Reachable Query
17:         if FActive = $\phi$ ∨ BActive = $\phi$ then
18:             return ( Not-Reachable )
19:         end if
20:         ▷ Case II: Reachable Query
21:         for all $v \in$ Processed do
22:             if VisitF ($v$) ∧ VisitB ($v$) then
23:                 Reachable ← true
24:                 newValue ← estimateApprox($v$)
25:                 safeApprox ←
26:                     $f_{approx}$ ( newValue, safeApprox )
27:             end if
28:         end for all
29:         if Reachable then
30:             Prediction ←
31:                 |FActive| > |BActive|
32:                     ? Backward : Forward
33:             break
34:         end if
35:     end while
36:     ▷ Phase 2
37:     if Prediction = Forward then
38:         ▷ Initialize destination $d$ vertex value
39:         $d.value$ = safeApprox
40:         ▷ Continue iterating: forward direction only
41:         while FActive ≠ $\phi$ do
42:             FActive ← Process ( FActive, $d$, $G$ )
43:         end while
44:         return ( Reachable, d.value )
45:     else ▷ Prediction is Backward
46:         ▷ Initialize source $s$ vertex value
47:         $s.value$ = safeApprox
48:         ▷ Continue iterating: backward direction only
49:         while BActive ≠ $\phi$ do
50:             BActive ← Process ( BActive, $s$, $\widehat{G}$ )
51:         end while
52:         return ( Reachable, s.value )
53:     end if
54: end function
```

execution in the non-predicted direction is discontinued. Note that all the processing performed in Phase 1 for the predicted direction is not wasted as computation continues from where it was for the predicted direction. At the start of Phase 2, if the predicted direction is forward the initial value for *destination* vertex d is set to **safeApprox** produced by Phase 1 and if the predicted direction is backward the initial value for the *source* vertex s is set to **safeApprox**.

Algorithm 2 summarizes the two-phase algorithm. The iterative loop (lines 7–35) representing Phase 1 processes active vertices and identifies active vertices for the next iteration. Phase 1 terminates under two conditions. First is when the query is found to be *non-reachable* because the active set in one of the directions becomes empty and thus the algorithm terminates (see lines 16–19). Second is when the query is found to be *reachable* in which case safe approximation is computed and direction for Phase 2 is predicted (see lines 20–34). The Phase 2 (lines 36–53) simply continues processing in the predicted direction, using the safe approximation, and terminates when the algorithm converges. During processing of active vertices in Phase 1 pruning is always off while in Phase 2 pruning is always on.

Note that the proposed algorithm satisfied all three requirements. Our approach handles both non-reachable and reachable queries (RQ1). For non-reachable queries our execution time is expected to be close to the faster direction time which is much smaller than the slower direction time. For reachable queries since Phase 1 is fast, Phase 2 is highly optimized as our algorithm accurately predicts the faster direction (RQ2) and maximizes the use of pruning by ensuring that it is enabled right from the start of Phase 2 (RQ3).

Applicability of PnP. The **PnP** two phase algorithm minimizes computations by limiting propagation of values via direction selection and safe pruning. We further understand how direction selection and pruning can be applied to a wide variety of graph algorithms. Graph algorithms are typically convergence based iterative algorithms wherein vertex values propagate as they change across iterations. These propagations happen across the structure of the input graph, and hence, they can be viewed as occurring in certain pattern or direction. At an elementary level, propagation of a vertex value occurs in the "outward" direction through out-neighbors of the vertex; for example, in Algorithm 1, the out-neighbors of vertices get processed (line 14) as values propagate across the graph. However, an important characteristic of point-to-point queries is the two special vertices (a source and a destination) that concretely define an expected direction for propagation: *forward* direction from source to destination. **PnP** further extracts the hidden reverse direction to leverage the disparity in propagation and limits overall computations via pruning. Path based algorithms naturally fit this class of point queries where values

G	Algorithm	WP	SP	NP	BFS	Vertices Visited
TTW	2Phase	0.1438s (99.1%)	0.1447s (99.1%)	0.1397s (97.1%)	0.1007s (89.2%)	0.1767%
	sPr:FastNR	0.0237s	0.0188s	0.0283s	0.0072s	0.0000029%
	sPr:SlowNR	13.0250s	14.6090s	3.8500s	0.8727s	90.62%
LJ	2Phase	0.0191s (96.6%)	0.0188s (77.8%)	0.0271s (91.0%)	0.0188s (77.5%)	0.38 %
	sPr:FastNR	0.0009s	0.0011s	0.0064s	0.0011s	0.0299 %
	sPr:SlowNR	0.5337s	0.8200s	0.2365s	0.0796s	88.95 %
TT	2Phase	0.1744s (99.0%)	0.1991s (99.7%)	0.2672s (96.9%)	0.1486s (88.8%)	0.47%
	sPr:FastNR	0.0212s	0.1392s	0.0562s	0.0200s	0.0000024%
	sPr:SlowNR	15.3860s	17.6580s	6.8800s	1.1686s	84.60%
PK	2Phase	0.0084s (94.3%)	0.0085s (94.8%)	0.0122s (94.1%)	0.0085s (71.6%)	0.0688%
	sPr:FastNR	0.0004s	0.0005s	0.0023s	0.0004s	0.000066%
	sPr:SlowNR	0.1410s	0.1553s	0.1708s	0.0289s	80.77%

Table 7. NR Queries 2Ph vs. sPr: Average Execution Times (seconds); and % of Vertices Visited.

are expected to be propagated from source to destination. For general algorithms like PageRank, every vertex acts like a source; thus, it is difficult to deduce a single direction of flow of values that can be leveraged by **PnP**.

On the other hand, pruning of value propagation occurs when we know (a) what to prune; and, (b) how to prune it.

— *What to prune?* While **PnP** prunes value propagations (or edge computations) in a broader sense, the semantics of each graph algorithm needs to be carefully analyzed to identify the exact propagation paths across which values will never be transferred. These semantics can be captured by characterizing the aggregation function used to compute vertex values. The most common aggregation functions used across graph algorithms are shown in Table 8. Since selection based aggregation functions (*min*, *max*, *or*) effectively select values coming from a single incoming path to a given vertex, **PnP** can safely prune values coming from other incoming paths to a vertex, hence supporting several graph algorithms, some of which are listed in Table 8. Complete aggregations (*sum*, *product*) on the other hand combine values coming from multiple incoming edges into a single value. This means the value contributions from individual incoming paths cannot be discarded throughout the computation, and hence, **PnP** does not prune value propagations but instead only performs direction selection. In our experiments (§4), we use NumPaths as an example for complete aggregation to show that **PnP** is very useful even without pruning.

Aggregation	Type	Graph Algorithms
min, *max*, *or*	Selection	Shortest Paths, Widest Paths, Connected Components, Reachability Minimum Spanning Tree Betweenness Centrality
sum, *product*	Complete	NumPaths, PageRank, SpMV, Belief Propagation

Table 8. Applicability of **PnP**.

— *How to prune?* Once we have identified the propagation paths to prune, we rely on the algorithmic semantics to perform pruning. Vertex values for path based algorithms that rely on selection functions often progress in a monotonic fashion, i.e., subsequent values of vertices are either non-increasing (e.g., shortest paths) or non-decreasing (e.g., widest paths). For such algorithms, **PnP** monitors the destination vertex's values and performs numerical comparison (≥, ≤) to safely prune out propagations that can never contribute to final result. For algorithms beyond monotonic convergence (e.g., PageRank), other algorithm-specific pruning conditions can be formulated by the user.

4 Evaluation of PnP Two-Phase

We evaluate the two-phase algorithm with four input graphs and four graph analytics benchmarks. We use four input graphs from Table 3. Four types of queries are considered – Widest Path (WP), Shortest Path (SP), Number of Paths (NP), and Breadth First Search (BFS). We first evaluate the two-phase algorithm for non-reachable queries and then for reachable queries. The algorithms compared are as follows:

- 2Phase (2Ph) – our two-phase algorithm (from §3); &
- sPr – simple Pruning algorithm that can be run in forward or backward direction (from §2).

4.1 Evaluation for Non-Reachable (NR) Queries

The execution times for 2Phase as well as sPr in forward and backward directions for all non-reachable queries are shown in the scatter plots of Figure 4. As we can see, for vast majority of queries the execution time of 2Phase algorithm is very close to the time for the faster direction which is significantly smaller that the time in the slower direction.

The average times across all queries for sPr in the faster direction (sPr:FastNR) and slower direction (sPr:SlowNR) as well as 2Phase algorithm can be found in Table 7. The effectiveness of 2Phase algorithm is computed as

$$\frac{sPr : SlowNR - 2Phase}{sPr : SlowNR - sPr : FastNR} \times 100$$

which computes *actual benefit* of two-phase as a percentage of *available benefit* – this number is shown in parenthesis in Table 7. This number is often over 90%. The last column in the table (**Vertices Visited**) indicates the fraction of vertices in the entire graph that are visited by each algorithm. The numbers for sPr:FastNR and sPr:SlowNR confirm that a tiny

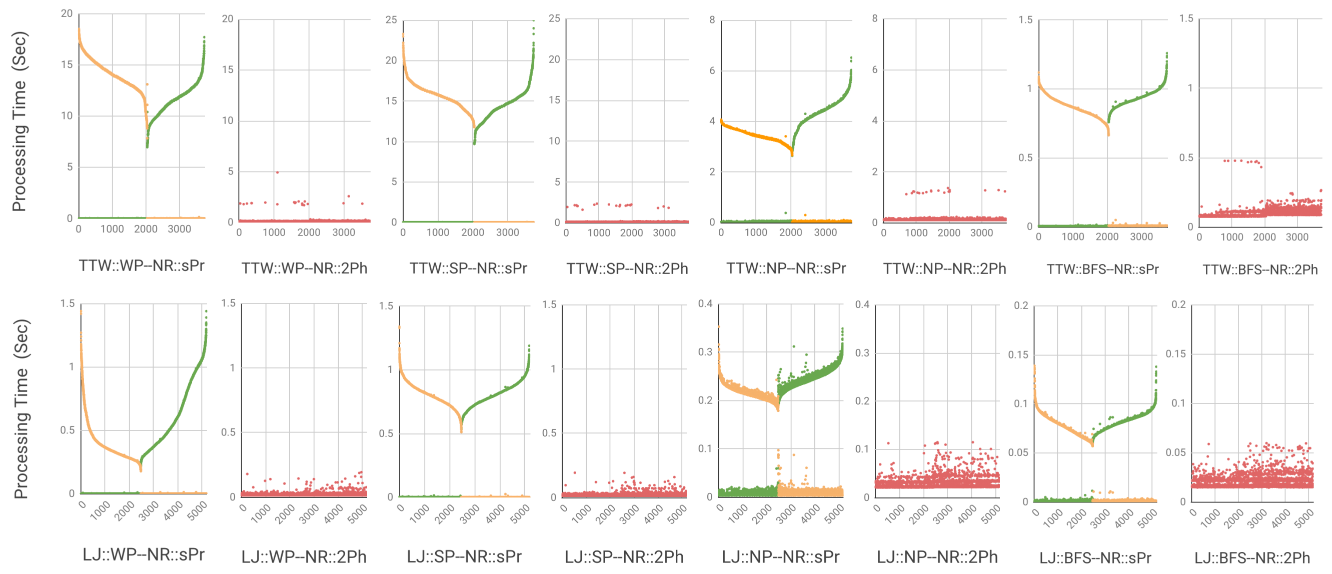

Figure 4. Execution Times of NR Queries: [Left] sPr Forward (Green) & sPr Backward (Gold); and [Right] 2Ph (Red).

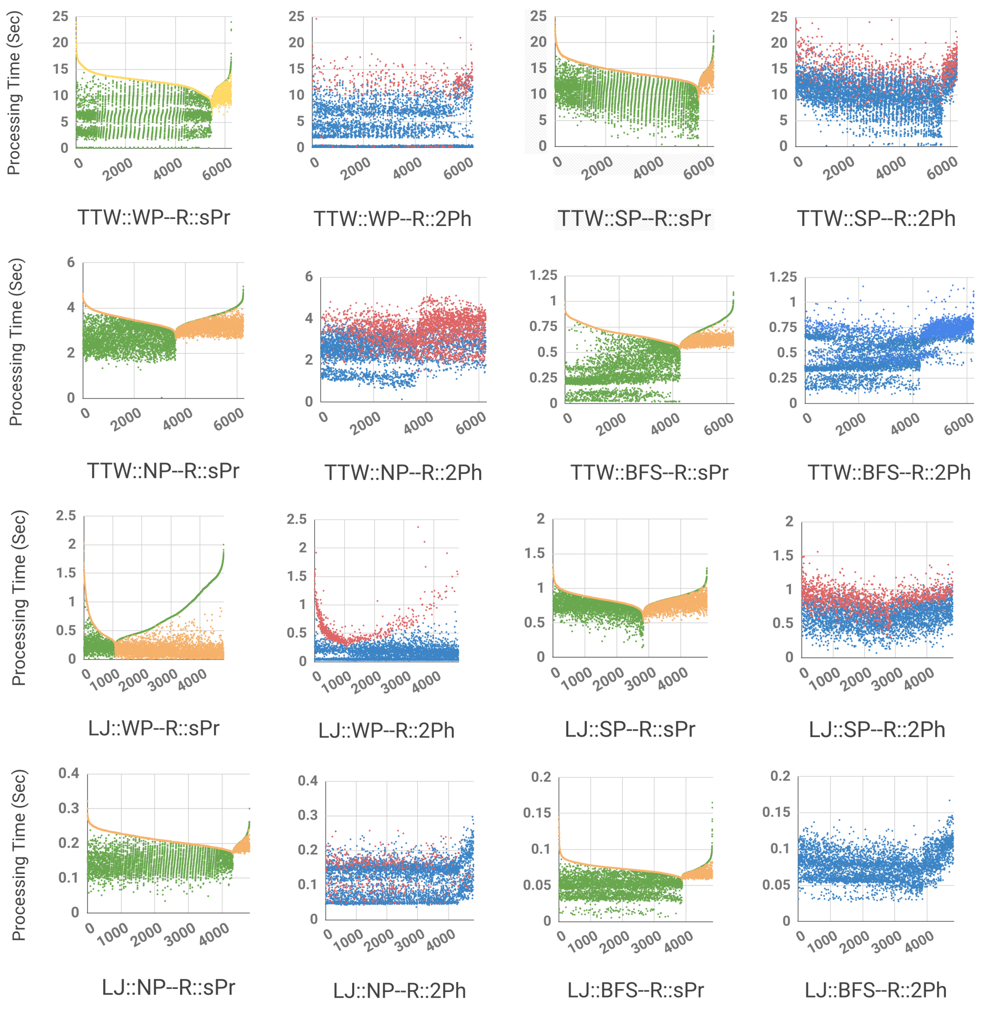

Figure 5. Execution Times of R Queries: [Left] sPr Forward (Green) & sPr Backward (Gold); and [Right] 2Ph Direction Correctly Predicted (Blue) & 2Ph Direction Mispredicted (Red).

G	Algorithm	WP	SP	NP	BFS
TTW	2Ph	**3.5116s**	10.827s	2.9134s	0.5396s
	2Ph100%	**3.2826s**	10.466s	2.6813s	–
	sPr:FastR	5.9797s	10.038s	2.3898s	0.3585s
	sPr:SlowR	12.0750s	12.793s	3.1887s	0.6007s
LJ	2Ph	0.1998s	0.6928s	**0.1179s**	0.0781s
	2Ph100%	**0.1572s**	0.6543s	**0.1169s**	–
	sPr:FastR	0.1832s	0.6190s	0.1234s	0.0369s
	sPr:SlowR	0.7569s	0.8168s	0.1575s	0.0560s
TT	2Ph	**4.3782s**	16.727s	4.7800s	0.8338s
	2Ph100%	**3.8370s**	15.049s	4.5825s	–
	sPr:FastR	7.7483s	14.145s	4.0473s	0.5214s
	sPr:SlowR	14.6030s	19.041s	5.2159s	0.7136s
PK	2Ph	0.0705s	0.1392s	0.0980s	0.0342s
	2Ph100%	0.0631s	0.1361s	0.0968s	–
	sPr:FastR	0.0526s	0.1174s	0.0613s	0.0141s
	sPr:SlowR	0.1705s	0.1492s	0.0789s	0.0195s

Table 9. R Queries: Avg. Execution Time Per Query (secs).

fraction of the vertices are visited (< 0.03%) in the fast direction while vast majority of vertices are visited (80-90%) in the slower direction. Finally, two-phase algorithm visits less than 0.5% percent of the vertices explaining its effectiveness for non-reachable queries.

4.2 Evaluation for Reachable Queries

Let us analyze the performance of the two-phase algorithm for reachable queries. We again compare its performance with that of the limits of performance of the sPr algorithm (i.e., in faster and slower directions for all queries). The scatter plots for reachable queries are shown in Figure 5.

Average execution times across all reachable queries for algorithms sPr and 2Phase are given in Table 9. As we see in most cases execution times of algorithms are related as follows: sPr:FastR < 2Phase < sPr:SlowR. This is to be expected as sPr:FastR is in a sense ideal time where overhead of prediction is nil and prediction rate is 100%. In comparison 2Phase algorithm involves overhead of direction prediction and has less than perfect prediction rate. However, as we can see 2Phase is frequently far closer to sPr:FastR than sPr:SlowR. This indicates that 2Phase is highly effective. To further demonstrate its effectiveness, we also present the average execution time 2Phase100% which is computed assuming perfect 100% prediction rate. We can see that 2Phase is only slightly greater than 2Phase100%. Finally, it should be noted that in some cases 2Phase < sPr:FastR (WP on TTW, NP on LJ) or at least 2Phase100% < sPr:FastR (WP on LJ). This is because the 2Phase pruning strategy significantly outperforms the pruning carried out by sPr and thus more than offsets the cost of prediction. Note that for BFS no times for 2Phase100% are provided as BFS terminates at the end of Phase 1. Next we further analyze prediction and pruning.

Prediction effectiveness of 2Phase algorithm is analyzed in Table 10. The prediction rates (PR) of the 2Phase algorithm are presented – on an average the prediction rates exceed

G	Pred	WP	SP	NP
TTW	PR	92.74%	87.69%	56.51%
	ΔP	11.64s	5.67s	0.90s
	ΔMP	3.16s	2.93s	0.53s
LJ	PR	86.35%	71.67%	90.17%
	ΔP	0.61s	0.22s	0.04s
	ΔMP	0.31s	0.14s	0.01s
TT	PR	88.32%	57.24%	58.39%
	ΔP	12.75s	4.84s	0.45s
	ΔMP	4.63s	3.92s	0.48s
PK	PR	87.99%	87.80%	81.67%
	ΔP	0.13s	0.06s	0.015s
	ΔMP	0.06s	0.026s	0.003s

Table 10. 2Ph Prediction Effectiveness: (PR) Prediction Rate of 2Phase Algorithm; and Difference Between Average Execution Times (seconds) in Faster and Slower Directions for Predicted Queries (ΔP); and Mispredicted Queries (ΔMP). **BFS** is omitted as it does not require Phase 2.

G	Algorithm	WP	SP	NP
TTW	2Ph	28%	1.6%	6.1%
	2Ph100%	30%	1.7%	6.1%
	sPr:FastR	54%	31%	64%
	sPr:SlowR	45%	36%	74%
LJ	2Ph	21%	3.7%	22%
	2Ph100%	22%	3.1%	22%
	sPr:FastR	32%	8.3%	45%
	sPr:SlowR	42%	18%	86%
TT	2Ph	16%	1.2%	3.3%
	2Ph100%	14%	1.0%	2.6%
	sPr:FastR	29%	17%	28%
	sPr:SlowR	41%	27%	49%
PK	2Ph	2.6%	1.1%	2.9%
	2Ph100%	≈ 0%	≈ 0%	≈ 0%
	sPr:FastR	≈ 0%	≈ 0%	≈ 0%
	sPr:SlowR	41%	25%	49%

Table 11. R Queries: % of Execution Time for which Pruning is Inactive. **BFS** is omitted because it does not require Phase 2 as it terminates at the end of Phase 2.

80%. For the two cases where 2Phase < sPr:FastR we can see that the prediction rates exceed 90% (92.74% for WP on TTW; 90.17% for NP on LJ). Additional data in Table 10 shows that for queries where predictions are correct, on average, the difference in execution times in two directions (ΔP) is typically greater than for queries where misprediction occurs (ΔMP). That is, benefit of correct predictions is higher than the loss due to incorrect predictions.

Pruning effectiveness of 2Phase algorithm is analyzed in Table 11. We present the percentage of execution time over which pruning is not enabled – lower numbers are better. For 2Phase algorithm this time is the percentage of execution time spent in Phase 1. For sPr algorithm we found this time by noting the point at which the first approximation of query

G	Algorithm	WP	SP	NP	BFS
TTW	2Ph	5.82m	22.17m	27.86m	6.93m
	2Ph100%	5.17m	21.80m	15.27m	–
	sPr:FastR	14.65m	32.46m	54.13m	11.54m
	sPr:SlowR	51.16m	55.34m	48.06m	13.46m
LJ	2Ph	1.58m	6.12m	0.32m	0.80m
	2Ph100%	1.24m	6.31m	0.29m	–
	sPr:FastR	1.98m	6.85m	3.50m	0.85m
	sPr:SlowR	8.47m	12.15m	5.72m	2.13m
TT	2Ph	7.56m	30.56m	42.13m	7.51m
	2Ph100%	6.27m	39.07m	42.13m	–
	sPr:FastR	16.74m	59.87m	54.10m	12.04m
	sPr:SlowR	60.03m	50.34m	63.87m	17.77m
PK	2Ph	0.61m	1.38m	0.39m	0.28m
	2Ph100%	0.56m	1.37m	0.77m	–
	sPr:FastR	0.81m	1.70m	2.57m	0.37m
	sPr:SlowR	2.70m	2.99m	3.46m	0.67m

Table 12. R Queries: Average Active Vertex Count Per Query (in millions).

result is generated for use in pruning during remainder of the execution. From the results we can see that for the 2Phase algorithm this time is often significantly smaller than for sPr:FastR algorithm. That is, 2Phase pruning is substantially better than simple pruning performed by sPr.

Tables 12 and 13 present the work performed in terms of total number of active vertices encountered and number of iterations for convergence. As we can see, the number of active vertices is significantly smaller for 2Phase in comparison to sPr:FastR. This reduction is the highest for NP because the SafeApprox is computed by multiplying the NP values in forward and backward direction causing pruning to be highly effective. This is another indicator of the enhanced pruning strategy of 2Phase algorithm being significantly superior than that of sPr. On the other hand, the average number of iterations for 2Phase and sPr:FastR are fairly close. Note that even though the 2Phase algorithm for BFS terminates at end of Phase 1, its vertex computations count is consistently lower than that for sPr:FastR indicating that the bidirectional traversal is more effective that unidirectional traversal. Finally, occasionally 2Phase performs fewer active vertices than 2Phase100% (e.g., SP on LJ). This is because the runtime cost depends upon additional factors (e.g., number of edges associated with active vertices, cache misses etc.), i.e. the direction in which fewer active vertices are processed can have higher execution time.

Beamer's Bidirectional BFS [4] vs. *Two-Phase PnP.* Recently bidirectional BFS was proposed by Beamer that switches directions at iteration boundaries to minimize the work performed – the frontier sizes in two directions are compared to select the cheaper direction for the next iteration. Although this is an effective algorithm, **PnP** relies upon direction selection over bidirectional search. First, **PnP** is general which solves problems besides BFS while Beamer's algorithm only applies to BFS. Second, due to **PnP**'s aggressive pruning, the number of iterations in the two-phase algorithm are greatly reduced and this limits the potential benefits of bidirectional approach. Table 14 shows that the number of iterations of two-phase are much smaller than for no-pruning (noPr) scenario considered in bidirectional BFS. Thus, bidirectional approach is not expected to yield significant additional benefit in the presence of pruning.

G	Algorithm	WP	SP	NP	BFS
TTW	2Ph	5	9	5	4
	2Ph100%	4	9	4	–
	sPr:FastR	5	9	5	4
	sPr:SlowR	16	12	5	4
LJ	2Ph	10	28	4	6
	2Ph100%	6	27	4	–
	sPr:FastR	7	27	7	6
	sPr:SlowR	47	29	7	6
TT	2Ph	5	9	5	4
	2Ph100%	4	10	5	–
	sPr:FastR	5	11	5	4
	sPr:SlowR	17	10	5	4
PK	2Ph	8	14	4	5
	2Ph100%	6	14	4	–
	sPr:FastR	6	14	7	5
	sPr:SlowR	20	16	7	5

Table 13. R Queries: Average Number of Iterations Per Query (rounded).

G	Algorithm	WP	SP	NP	BFS
		#Iter.	#Iter.	#Iter.	#Iter.
TTW	2Ph	5	9	5	4
	noPr:FastR	20	21	23	19
	noPr:SlowR	21	21	27	20
LJ	2Ph	10	28	4	6
	noPr:FastR	20	30	15	14
	noPr:SlowR	46	32	16	15

Table 14. No-pruning (nPr) vs. Pruning in Two-Phase.

4.3 Quegel vs. PnP

Finally we compare the performance of **PnP** with Quegel, that is aimed at point-to-point iterative graph analytics. Table 15 shows the average execution times of **PnP** for 50 queries of each of four kinds on a single 8-core machine, and the average relative speedups achieved by **PnP** over Quegel on 1 and 4 machines (8-cores per machine). Quegel's optimization that combines messages with the same destination vertex is turned on, and results are shown for Quegel's bidirectional BFS (BiBFS) as well as unidirectional BFS. On an average across all types of queries, **PnP** on a single machine outperforms Quegel on four (one) machines by 8.2× to 3116× (7× to 1517×). Furthermore, it was interesting to observe that Quegel's BFS performed better than it's BiBFS in few cases; nevertheless our prediction and pruning strategies allowed **PnP** to greatly outperform both Quegel's BiBFS and BFS.

Q	G	PnP :: 1 machine				Quegel :: 1 machine					Quegel :: 4 machines				
		WP	SP	NP	BFS	WP	SP	NP	BiBFS	BFS	WP	SP	NP	BiBFS	BFS
FwdNR	LJ	0.020s	0.020s	0.037s	0.030s	12.9×	13.1×	7.02×	11.0×	8.63×	14.9×	15.1×	8.20×	12.0×	10.2×
	PK	0.007s	0.009s	0.011s	0.013s	62.5×	45.2×	23.0×	22.9×	31.0×	58.2×	41.9×	97.1×	24.7×	28.0×
BwdNR	LJ	0.027s	0.028s	0.045s	0.034s	679.4×	644.5×	877.3×	24.5×	521.1×	364.8×	344.2×	427.5×	19.2×	290.8×
	PK	0.007s	0.007s	0.012s	0.013s	1082.9×	1050.9×	1516.8×	22.8×	611.0×	618.7×	596.1×	3116.1×	23.4×	297.4×
FwdR	LJ	0.035s	0.198s	0.200s	0.136s	557.9×	97.7×	192.0×	17.5×	115.6×	320.8×	56.5×	98.4×	12.0×	67.5×
	PK	0.013s	0.081s	0.151s	0.054s	630.3×	99.5×	114.7×	22.3×	129.6×	353.9×	55.2×	229.0×	14.5×	64.4×
BwdR	LJ	0.042s	0.169s	0.214s	0.152s	438.0×	110.6×	169.7×	19.3×	101.5×	262.8×	58.3×	83.0×	13.5×	61.2×
	PK	0.012s	0.071s	0.148s	0.055s	629.6×	113.2×	112.3×	23.8×	122.2×	356.6×	63.1×	233.9×	15.5×	61.3×

Table 15. PnP average execution times in seconds for 50 queries of each kind on one 8-core 32 GB machine; and **PnP** average speedups over Quegel for the same queries on one and four machines.

5 Related Work

Graph Databases and Query Systems. The work closely related to ours is Quegel [41]. However, as discussed earlier, it relies upon offline Hub^2 computation that is limited to shortest path queries on unweighted graphs and does not allow graphs to change between queries. All these problems are addressed by **PnP** using dynamic pruning and dynamic direction prediction. Quegel also supports another scenario where on a distributed system a batch of queries are simultaneously solved by efficiently sharing memory and computing resources among the queries. This is different from the scenario we consider – solving a stream of queries on a single machine, and answering each query as quickly as possible. In contrast while batching improves throughput, it does not improve latency of query responses. Moreover, the batching algorithm also relies on Hub^2 pre-computation. Note that our technique can benefit from connected components precomputation but we prefer dynamic techniques to avoid disadvantages of precomputation. There are works that improve performance of specific algorithms (e.g., delta stepping for SSSP [23]); however, our goal is to develop optimizations that apply to multiple iterative graph algorithms.

There has been a great deal of work on graph based query languages (e.g., Gremlin [29]) and query support in graph databases (e.g., Neo4J and DEX [2, 9, 20]) 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. Their strength lies in their ability to program wide range of queries. They are more suitable neighborhood queries [22, 27, 28, 38] including query decomposition and incremental processing devoted to pattern matching [40, 42]. In [27] authors develop algorithms for efficiently answering k-nearest neighbor queries (k-NN) that prunes the search to limit the graph that is explored. In [38] authors develop a fast neighborhood graph search algorithm using a new data structure called the bridge graph constructed from a large number of bridge vectors. In [22] a compressed representation of social networks is proposed to facilitate computation of neighbor queries. NScale [28] is another system for neighborhood-centric analytics on large graphs including analysis tasks such as ego network analysis, social circles, personalized recommendations, link prediction, influence cascades, and motif [24] counting. Although GraphX [11] supports both kinds of graph operators (i.e., neighborhood aggregation as well as join and structural operators) and can be used for iterative algorithms, it does not support iterative point-to-point queries.

Graph Processing Frameworks. There are a number of single machine shared-memory frameworks [1, 15, 26, 31]. *Ligra* [31] provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal and we build **PnP** using Ligra. [26] presents a shared-memory based implementations of these DSLs on a generalized *Galois* system and compares its performance with the original implementations. These frameworks are based on the Bulk Synchronous Parallel (BSP) [33] model. *GRACE* [39], a shared memory graph processing system, uses message passing and provides asynchronous execution. To efficiently process large graphs our prior work has employed Graph Reduction [15] and built a system on top of *Galois*. On a single machine large graphs may not fit in memory. Therefore other methods have been proposed for processing extremely large graphs. For a single multicore machine a number of out-of-core processing systems have been recently proposed (GraphChi [17], X-Stream [30], GridGraph [45], DynamicShards [37], Turbograph [12], Flashgraph [44], Bishard [25]). Alternately distributed systems that combine memories of multiple machines to handle large graphs can be used (Pregel [21], PowerLyra [7], PowerGraph [10], GraphLab [19], ASPIRE [34], CoRAL [35]). Asynchronous algorithms are more capable of tolerating communication latencies of distributed systems [13, 34, 37, 43].

6 Conclusions

We have developed **PnP** that incorporates a two-phase algorithm for evaluating iterative point-to-point queries involving a single source and destination vertex pair. The algorithm derives its efficiency from selecting the faster direction for evaluating the query and pruning the computation to achieve early termination. Our solution is applicable to streaming graphs. Finally, **PnP** substantially outperforms Quegel, the only framework prior for computing point-to-point queries.

Acknowledgments

This work is supported by NSF grants CCF-1524852 and CCF-1813173 to University of California Riverside.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis* (SC), pages 1-11, 2010.

[2] A.B. Ammar. Query Optimization Techniques in Graph Databases. In *International Journal of Database Management Systems*, Vol.8, No.4, August 2016.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD), pages 44-54, 2006.

[4] S. Beamer, K. Asanovic, and D. Patterson. Direction-Optimizing Breadth-First Search. In *ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis* (SC), 10 pages, November 2012.

[5] M. Cha, H. Haddadi, F. Benevenuto, and P.K. Gummadi. Measuring user influence in twitter: The million follower fallacy. *International AAAI Conference on Web and Social Media* (ICWSM), 10(10-17):30, 2010.

[6] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. L. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 131-142, 2003.

[7] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *European Conference on Computer Systems* (EuroSys), Article 1, 15 pages, 2015.

[8] H. Cui, J. Cipar, Q. Ho, J.K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G.R. Ganger, P.B. Gibbons, G.A. Gibson, and E.P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX Annual Technical Conf.* (ATC), pages 37-48, 2014.

[9] Developers. Neo4J. *Graph NoSQL Database*, 2012.

[10] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symp. on Operating Systems Design and Implementation* (OSDI), pages 17-30, October 2012.

[11] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *USENIX Symp. on Operating Systems Design and Implementation* (OSDI), pages 599-613, 2014.

[12] W-S. Han, S. Lee, K. Park, J-H. Lee, and M-S. Kim, and J. Kim and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD), pages 77-85, 2013.

[13] A.F. Harshvardhan, N. M. Amato, and L. Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 27-38, 2014.

[14] R. Jin, N. Ruan, B. You, and H. Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. CoRR, abs/1305.0507, 2013.

[15] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu. Efficient Processing of Large Graphs via Input Reduction. In *ACM International Symposium on High-Performance Parallel and Distributed Computing* (HPDC), pages 245-257, May-June 2016.

[16] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591-600, 2010.

[17] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi : Large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 31-46, 2012.

[18] J. Leskovec. Stanford large network dataset collection. *http://snap.stanford.edu/data/index.html*, 2011.

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716-727.

[20] P. Macko, D. Margo, and M. Seltzer. Performance Introspection of Graph Databases. In *ACM International Systems and Storage Conferences* (SYSTOR), 10 pages, 2013.

[21] G. Malewicz, M.H. Austern, A.J.C Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*, pages 135-146, 2010.

[22] H. Maserrat and J. Pie. Neighbor Query Friendly Compression of Social Networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD), 9 pages, 2010.

[23] U. Meyer and P. Sanders. Δ-Stepping: A Parallelizable Shortest Path Algorithm. In *Journal of Algorithms*, 49:114-152, 2003.

[24] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298.5594, pages 824-827, 2002.

[25] K. Najeebullah, K. U. Khan, W. Nawaz and Y-K. Lee. BiShard Parallel Processor: A Disk-Based Processing Engine for Billion-Scale Graphs. In *International Journal of Multimedia and Ubiquitous Engineering*, 9(2):199-212, 2014.

[26] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles* (SOSP), pages 456-471, 2013.

[27] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. k-Nearest Neighbors in Uncertain Graphs. In *Proceedings of the VLDB*, VLDB Endowment, pages 997-1008, 2010.

[28] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Analytics on Large Graphs. In *International Conference on Very Large Data Bases* (VLDB), VLDB Endowment, Vol. 7, No. 13, pages 1673-1676, 2014.

[29] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *15th Symposium on Database Programming Languages* (DBPL), pages 1-10, 2015.

[30] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *24th ACM Symposium on Operating Systems Principles* (SOSP), pages 472-488, 2013.

[31] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 135-146, 2013.

[32] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pages 1-6, 2012.

[33] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM* (CACM), 33(8):103-111, 1990.

[34] K. Vora, S-C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM. In *SIGPLAN International Conf. on Object Oriented Programming Systems, Languages and Applications* (OOPSLA), pages 861-878, October 2014.

[35] K. Vora, C. Tian, R. Gupta, and Z. Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 223-236, April 2017.

[36] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 237-251, April 2017.

[37] K. Vora, G. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Distributive Disk-based Graph Algorithms. *USENIX Annual Technical Conference* (ATC), pages 507-522, June 2016.

[38] J. Wang, J. Wang, G. Zeng, R. Gan, S. Li, and B. Guo. "Fast Neighborhood Graph Search using Cartesian Concatenation," In *International Conference on Computer Vision* (ICCV), pages 2128-2135, 2013.

[39] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *Conference on Innovative Data Systems Research* (CIDR), pages 3-6, 2013.

[40] F. Wenfei , L. Jianzhong, L. Jizhou , T. Zijing , W. Xin and W. Yinghui. Incremental Graph Pattern Matching. In *ACM SIGMOD International Conference on Management of Data*, pages 925-936, 2011.

[41] D. Yan, J. Cheng, M.T. Ozsu, F. Yang, Y. Lu, J.C.S. Lui, Q. Zheng and W. Ng. A General-Purpose Query-Centric Framework for Querying Big Graphs. In *Proceedings of the VLDB Endowment*, Vol. 9, No. 7, pages 564-575, 2016.

[42] S. Yang, X. Yan, B. Zong and A. Khan. Towards effective partition management for large graphs. In *ACM SIGMOD International Conference on Management of Data*, pages 517-528, 2012.

[43] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 194-204, 2015.

[44] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies* (FAST), pages 45-58, 2015.

[45] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX Annual Technical Conference* (ATC), pages 375-386, 2015.