MITra: A Framework for Multi-Instance Graph Traversal

Jia Li
Edinburgh Research Centre, Huawei
jiali11@huawei.com

Wenyue Zhao
University of Edinburgh
wenyue.zhao@ed.ac.uk

Nikos Ntarmos
Edinburgh Research Centre, Huawei
nikos.ntarmos@huawei.com

Yang Cao
University of Edinburgh
yang.cao@ed.ac.uk

Peter Buneman
University of Edinburgh
peter.buneman@ed.ac.uk

ABSTRACT

This paper presents MITra, a framework for composing multi-instance graph algorithms that traverse from multiple source vertices simultaneously over a single thread. Underlying MITra is a model of multi-instance traversal that uniformly captures traversal sharing across instances. Based on this, MITra provides a programming model that allows users to express traversals by declaring vertex ranks and specify computation logic via an edge function. It synthesizes multi-instance traversal algorithms from declared vertex ranks and edge functions adopted from classic single-instance algorithms, automatically sharing computation across instances and benefiting from SIMD. We show that MITra can generate multi-instance algorithms provably better than existing ones, while being more expressive than traditional frameworks. In addition to the ease of programming, we experimentally verify that MITra is on average an order of magnitude faster than approaches based on existing frameworks for common graph algorithms, and is comparable to the state-of-the-art highly optimized one-off algorithms.

PVLDB Reference Format:
Jia Li, Wenyue Zhao, Nikos Ntarmos, Yang Cao, and Peter Buneman. MITra: A Framework for Multi-Instance Graph Traversal. PVLDB, 16(10): 2551 - 2564, 2023.
doi:10.14778/3603581.3603594

1 INTRODUCTION

Multi-instance processing (MIP) over large graphs is of increasing importance due to its applications in *e.g.,* social networks, bioinformatics, and web search [12, 24, 28, 29, 31, 39, 41, 48, 57]. It refers to the evaluation of multiple instances of the same query over the same graph but with different source vertices as input, *e.g.,* computing shortest paths from hundreds of vertices [13, 29, 57] simultaneously.

There are mainly two approaches to MIP. The first is highly optimized one-off MIP algorithms that compute answers to all sources simultaneously, by aligning and sharing computations among the sources. They are crafted for a specific graph computation, *e.g.,* MS-BFS [48] for BFS, and implement heavy algorithm-specific optimizations and heuristics. The other approach, referred to as serial algorithms, runs the instances in serial one by one, often by employing general-purpose graph frameworks [32, 35, 45, 55, 59, 61]. They are much easier to use due to the higher level framework interface, but tend to be less efficient due to the lack of sharing across instances.

This raises a question: can we have the best of both worlds? Can we have a framework with the interface of general-purpose frameworks while retaining the performance of one-off MIP algorithms?

MITra. To answer the question, we develop MITra, a framework for Multi-Instace graph Traversal computations.

(1) *Ease of programming*. MITra provides an edge-centric programming model that allows users to compose MIP algorithms that process a set of sources in one go with a single thread. With MITra, one can specify traversal logic by declaring numeric *ranks* for vertices and express query-dependent computation logic via an edge function, which is almost identical to those found in textbook single-instance graph algorithms. MITra synthesizes fully functional MIP algorithms from declared vertex ranks and edge functions.

(2) *Optimization*. MITra offers capabilities similar to those of one-off MIP algorithms for sharing computations among instances, but does so through an unified interface. This allows users to enjoy the optimizations of one-off MIP algorithms without the need to implement low-level subtleties, *e.g.,* aligning and sharing traversals, and proper application of SIMD (single-instruction-multiple-data).

(3) *Expressiveness*. With MITra, one can compose common graph traversal algorithms for their MIP cases. By virtue of its underlying algorithm model, MITra covers a wide range of graph algorithms, more expressive than traditional frameworks. Moreover, it even allows us to compose new algorithms that are particularly efficient for MIP, *e.g.,* a new MIP algorithm for graph reachability that is *provably* more efficient than existing ones on *each and every* graph.

(4) *Performance*. The ease of programming does not imply performance degradation compared to state-of-the-art one-off MIP algorithms and methods built upon general-purpose graph frameworks. For instance, over `Twitter` [8] with 4.16 million vertices and 1.47 billion edges and 256 sources, MITra is 33.56, 140.61, 6.7, 7.88 and 13.14 times faster than framework-based approaches for BFS, Reachability, Bellman-Ford, personalized PageRank and SpMV respectively; it is comparable to the highly engineered one-off MS-BFS [48] for BFS and even 1.73 times faster for graph reachability.

Frontier-ranking model. The foundation of MITra is a model called *frontier-ranking algorithm*. It provides a systematic approach to viewing and designing graph traversal algorithms, by separating traversal logic from computation logic. It expresses traversal logic via a numeric vertex property called ranks and compresses computation logic as edge functions. While computation logic is often

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603594

application/query dependent and varies a lot, traversal logic is much more regular and hence can be specified by declaring vertex ranks.

Frontier-ranking algorithms interpret traversal logic by organizing vertices into frontiers, determining the order in which the frontiers are explored, and identifying sharing opportunities. The entire process is guided by arithmetic operations over vertex ranks, hidden from users. This enables MITra to provide a rather simple, yet powerful, interface for composing multi-instance algorithms. Users only need to focus on computation logic from the view of an edge, in an edge function $f(e)$. MITra synthesizes full MIP algorithms that track and align traversal progresses of multiple instances via their vertex ranks, invoke $f(e)$ for the correct set of instances, and automatically extract sharing of edge accesses and invocations to $f(e)$ without impairing the correctness of each individual instance.

By varying vertex ranks, one can compose existing and new graph algorithms for their MIP cases. Indeed, MITra can express common graph computations, even those that are not expressible in existing graph frameworks, *e.g.,* Dijkstra. Moreover, the choice of ranks implies the level of sharing that one could get from MITra.

The use of numeric vertex properties also gives rise to MITra arithmetic operators, with which users can write MIP algorithms by adopting textbook single-instance algorithms with minimal efforts. MITra operators also benefit from *e.g.,* SIMD from modern CPUs.

Contribution. In summary, we make the following contributions:

- We propose a model for multi-instance graph algorithms. We show that it is effective for multi-instance computations and more expressive than existing graph computation models (Section 3).
- Based on the model, we develop MITra, a programming framework for multi-instance graph computations (Section 4). We present its interface and show how it synthesizes MIP algorithms. We also discuss its implementation and optimizations.
- We show that with MITra one can write multi-instance algorithms as simple as their single-instance counterparts (Section 5).
- We empirically verify that MITra outperforms approaches built on traditional graph frameworks by an order of magnitude, comparable to highly optimized one-off MIP algorithms (Section 6).

2 PRELIMINARIES

Graphs. A graph is denoted by $G(V, E, w)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges of G, w is a function that assigns each edge $e \in E$ a weight $w(e)$. As usual, we assume that there is an iterator that, given a vertex $v \in V$, returns all neighbors of v in G in some deterministic order, *e.g.,* increasing in vertex IDs.

Graph queries. We consider the below queries over graph G.

Graph search. We consider two graph search queries:

(a) BFS. A BFS query over G is specified by a vertex s called *source*; it returns a Breadth-First-Search tree [16] or the distance of each vertex from s when the weight $w(e)$ of each edge is 1 [45, 48].

(b) Reachability search. A graph reachability query [46] over G starts from a source vertex s and returns all vertices that are reachable from s in G; these can be found with a BFS starting from s [18].

Shortest path. Single-source shortest path (SSSP) queries find a shortest path from a source s to every vertex in an edge-weighted graph that is reachable from s. Popular algorithms for SSSP include, *e.g.,* Bellman-Ford [16], Dijkstra [16] and Δ-stepping [34].

Personalized PageRank (PPR) is the probability that a random walk in G from s terminates at t, which has been used to measure the bidirectional importance between s and t [36, 51, 52]. A PPR query asks, given a source s, the PPR value from s to every vertex in G.

Sparse matrix-vector multiplication (SpMV) multiplies the adjacency matrix A (transposed) of graph G with a vector x of values, one per vertex; it is used in Graph/Recurrent Neural Networks [40], Topic Search [25], and Belief Propagation [22]. It has been a common practice to use graph frameworks to compute Ax, by treating x as a 'virtual' vertex v_x and casting Ax as a graph traversal step from v_x in G [17, 27, 42, 47]. Given an x, an SpMV query computes Ax.

Multi-instance computation. Consider a graph G and graph query class Q, *e.g.,* reachability. A *multi-instance processing* (MIP) algorithm for Q computes, given a set of sources $s_1, \ldots, s_k$, the results to each source s_i, *e.g.,* vertices of G that are reachable from s_i.

In this paper, we consider MIP algorithms over a single thread, and study how to compute the answers to multiple sources in one go, in an interleaved manner. There are two existing approaches.

(1) Specialized one-off algorithms that exploit deep algorithm-specific optimizations, *e.g.,* MS-BFS [48] for multi-instance BFS with algorithm-specific optimization ranging from CPU register alignment to pre-fetching, and to algorithmic heuristics; similarly for MS-Dijkstra [57], a one-off Dijkstra variant for multi-instance SSSP.

(2) Serial algorithms that compute answers to each source serially, one after another, by employing convenient graph frameworks.

Our goal is to develop MITra, an MIP framework that provides the best of both worlds: (a) comparable performance to one-off algorithms and (b) ease of use and simplicity of graph frameworks.

3 MODELING MULTI-INSTANCE TRAVERSALS

Underlying MITra is a model of multi-instance graph computations, referred to as the *frontier-ranking model*. Below we first present the model (Section 3.1). We then justify its novelty by proving its effectiveness (Section 3.2) and expressive power (Section 3.3).

3.1 The Frontier-Ranking Model

Foundational to MITra is the frontier-ranking model that abstracts *one-off style* MIP graph algorithms. It models traversals by means of a *numeric* vertex property called *ranks*, expresses interleaving traversal logic across multiple instances via numeric operations on vertex ranks, and separates it from application specific computation logic encapsulated in *edge functions*. In doing so, it pushes housekeeping for multi-instance computations into numeric computations on ranks instead of imperative instructions in edge functions.

Single-instance frontier-ranking. For readability, we first present frontier-ranking algorithms for single-instance traversals.

Given source s of graph G, a single-instance frontier-ranking algorithm $\mathcal{A}$ traverses G in rounds starting from s. In each round a frontier F is explored and one or more new frontiers are generated. Frontiers are maintained in order via a structure denoted by $\mathbb{F}$.

More specifically, as shown in Fig. 1(a), $\mathcal{A}$ traverses G as follows:

(1) Initially, $\mathbb{F}$ consists of a single frontier for source s.

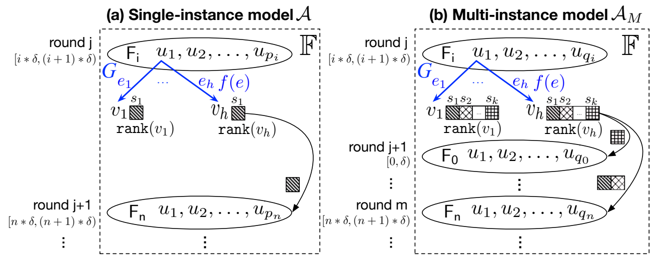

(a) Single-instance model $\mathcal{A}$
(b) Multi-instance model $\mathcal{A}_M$
round j
$[i*\delta,(i+1)*\delta)$
F_i $u_1, u_2, \ldots, u_{p_i}$
F_i $u_1, u_2, \ldots, u_{q_i}$
$\mathbb{F}$
G
e_1
e_h $f(e)$
s_1
$s_1 s_2$ s_k
v_1
v_h
rank(v_1)
rank(v_h)
round j+1
$[0,\delta)$
F_0 $u_1, u_2, \ldots, u_{q_0}$
round m
$[n*\delta,(n+1)*\delta)$
F_n $u_1, u_2, \ldots, u_{p_n}$
F_n $u_1, u_2, \ldots, u_{q_n}$

Figure 1: frontier-ranking algorithms

(2) For each round j, it explores the top frontier F from $\mathbb{F}$ for the current round, by exploring each vertex $u \in \mathsf{F}$, as follows:

  (a) for each edge $e = (u, v)$ in G, it reads in e, carries out some computation $f(e)$ on e, and updates the *rank* of v;

  (b) it then assigns v to frontier F' if the rank of v is *changed* in (a) and falls in the *domain* of F'; if such F' does not exist in $\mathbb{F}$, it creates one for v and adds it to $\mathbb{F}$.

(3) After exploring all vertices in frontier F, $\mathcal{A}$ fetches the next frontier F'' from $\mathbb{F}$ and moves to round $j+1$ to explore F''.

(4) The iteration terminates if no frontiers remain in $\mathbb{F}$.

Algorithm $\mathcal{A}$ provides the following parameters.

(a) Edge function $f(e)$ is the main function to express application-specific graph computation logic. Note that, when $\mathcal{A}$ visits an edge $e = (u, v)$, u must be a vertex in the current frontier. Function $f(e)$ updates vertex properties of v by propagating those of u according to the computation logic. For instance, for SSSP, $f(e)$ may update property $\texttt{ans}(v)$, the distance of v from the source vertex, by setting it to $\min(\texttt{ans}(u)+w(e), \texttt{ans}(v))$, where $w(e)$ is the weight of edge e.

(b) Rank $\texttt{rank}(v)$ is a reserved runtime property for each vertex in G. For each v of G, $\texttt{rank}(v)$ is a real number. Its definition specifies how $\texttt{rank}(v)$ is updated when $\mathcal{A}$ visits v via some edge $e = (u, v)$. Common rank definition includes the number of rounds (#round), vertex ID (vid), or query answers (*e.g.,* ans).

(c) Frontier width δ. Vertices are grouped into frontiers according to their ranks such that each frontier F covers vertices with ranks that fall into a range called the domain of F. We consider *equal-width* domains for frontiers in this work and denote by δ the width of a frontier: a frontier F with *index* r ($r \in \mathbb{N}$) has domain $[r \cdot \delta, (r+1) \cdot \delta)$. Hence, in step (2b), a vertex v with $\texttt{rank}(v)$ is assigned to frontier F with index r if $\texttt{rank}(v)/\delta \in [r, r+1)$.

(d) Structure $\mathbb{F}$. Note that there may be multiple unexplored frontiers during traversing. To this end, $\mathcal{A}$ organizes them in an appropriate structure $\mathbb{F}$, *e.g.,* a heap or a list. In each round, $\mathbb{F}$ pops out the "top" frontier for $\mathcal{A}$ to explore. For instance, when $\mathbb{F}$ is a list, frontiers are organized and explored in an FIFO (first-in-first-out) order according to when they are generated and put in $\mathbb{F}$ by $\mathcal{A}$.

Initially, $\mathbb{F}$ is empty; when $\mathcal{A}$ assigns a newly visited vertex v to frontiers in $\mathbb{F}$ and no frontier in $\mathbb{F}$ has index $\lfloor \texttt{rank}(v)/\delta \rfloor$, a new frontier F is then created for v with index $\lfloor \texttt{rank}(v)/\delta \rfloor$ and is added to $\mathbb{F}$.

Intuitively, (a) expresses graph *computation logic* from the view of an edge (blue in Fig. 1), while (b), (c) and (d) together capture and normalize the *traversal logic* by "ranking" vertices into frontiers according to their ranks (black in Fig. 1). This has two immediate benefits: (i) it expresses a larger class of algorithms than existing models (Table 1 shows some example algorithms; see more in Section 3.3); and (ii) traversal logic is encoded as numeric operations on rank instead of imperative instructions, *independent of* specific algorithms.

Table 1: Graph computations in frontier-ranking model

Traversals	$f(e)$[1]	$\texttt{rank}(v)$	δ	$\mathbb{F}$
BFS	bfs	**#round**: # of rounds in which v is visited.	1	list
Reachability	bfs	vid: ID of vertex $v \in V$ in data graph G.	1	list
Bellman-Ford	distance	**#round**: # of rounds in which v is visited.	1	list
Dijkstra	distance	vid: ID of vertex $v \in V$ in data graph G.	1	heap
Δ-stepping	distance	ans: tentative distance to the source; $\texttt{ans}(v) = \min(\texttt{ans}(u) + w(u,v), \texttt{ans}(v))$.	Δ	heap
PPR	pagerank	**#round**: # of rounds in which v is visited.	1	list
SpMV	spmv	**#round**: # of rounds in which v is visited.	1	list

[1] see detailed implementations of $f(e)$ in Section 4.1 and Section 5

Multi-instance frontier-ranking. We next show how the model captures one-off style MIP algorithms, referred to as *multi-instance frontier-ranking* algorithms. Similar to the single instance case, a multi-instance frontier-ranking algorithm $\mathcal{A}_M$ traverses G in rounds such that in each round a frontier is explored and new frontiers are generated for future rounds. However, multiple instances may be visiting the same vertices in a frontier simultaneously, which provides opportunities for computation sharing.

Consider a set S of k sources $s_1, \ldots, s_k$ of G. To compute the answers to each s_i, $\mathcal{A}_M$ extends the previous special case of single-instance frontier-ranking algorithms $\mathcal{A}$ as follows:

(1) The rank of each vertex u, *i.e.,* $\texttt{rank}(u)$, is populated to k ranks such that $\texttt{rank}(u, s_i)$ is the rank of u for source s_i, for each $i \in [1, k]$; similarly for other vertex properties.

(2) Each F maintains, for each vertex u in F, all sources that are currently visiting u when F is being explored. This is done via a structure track: $\texttt{track}(u)$ is a boolean array of length k such that $\texttt{track}(u, s_i)$ is *True* if and only if source s_i is visiting u.

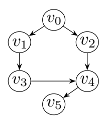
More specifically, as shown in Fig. 1(b), $\mathcal{A}_M$ traverses G in rounds, similar to $\mathcal{A}$. Initially, $\mathbb{F}$ consists of frontiers created for the sources according to their ranks. $\mathcal{A}_M$ starts the traversal with the top frontier of $\mathbb{F}$, similar to $\mathcal{A}$, but differs in steps (2a) and (2b):

(2a) for each edge $e = (u, v)$ in G, it carries out computation $f(e)$ on e for all sources s_i that are currently visiting v via e, *i.e.,* $\texttt{track}(u, s_i) =$ *True*, and updates $\texttt{rank}(v, s_i)$;

(2b) $\mathcal{A}_M$ then assigns v, *for each source* s_i, to some frontier F' according to $\texttt{rank}(v, s_i)$ if it is changed in step (2a): if there exists such F' in $\mathbb{F}$ whose domain covers $\texttt{rank}(v, s_i)$, it adds v to F' and sets $\texttt{track}(v, s_i)$ for F'; otherwise it creates F' with v for $\mathbb{F}$, with $\texttt{track}(v, s_i) =$ *True* and $\texttt{track}(v, s_j) =$ *False* for all $j \neq i$.

Sharing. Intuitively, $\mathcal{A}_M$ groups traversals at vertex v from multiple sources s_i as long as their ranks $\texttt{rank}(v, s_i)$ fall in the domain of the same frontier F' (step(2b)). In the subsequent rounds, whenever $\mathcal{A}_M$ explores vertex v in F', it groups edge accesses to $e = (v, w)$ for the sources s_i that visit v in the same round, *i.e.,* $\texttt{track}(v, s_i) =$ *True* (step (2a)). At this point the traversals from the sources are aligned and the computations for the sources can be therefore shared.

(a) Data graph



(b) Queries

$\{v_0, v_1, v_3\}$

(c) MITra-BFS/MS-BFS (Rank: #round)

	$\mathbb{F}$	$\underline{\mathsf{F}}$	edges
1	$\underline{\mathsf{F}_0}$	v_0:(1,0,0) v_1:(0,1,0) v_3:(0,0,1)	(v_0, v_1) (v_0, v_2) (v_1, v_3) (v_3, v_4)
2	$\underline{\mathsf{F}_1}$	v_1:(1,0,0) v_2:(1,0,0) v_3:(0,1,0) v_4:(0,0,1)	(v_1, v_3) (v_2, v_4) (v_3, v_4) (v_4, v_5)
3	$\underline{\mathsf{F}_2}$	v_3:(1,0,0) v_4:(1,1,0) v_5:(0,0,1)	(v_3, v_4) (v_4, v_5)
4	$\underline{\mathsf{F}_3}$	v_5:(1,1,0)	$\emptyset$

(d) MITra-RCH (Rank: vertex ID)

	$\mathbb{F}$	$\underline{\mathsf{F}}$	edges
1	$\underline{\mathsf{F}_0} \rightarrow \mathsf{F}_1 \rightarrow \mathsf{F}_3$	v_0:(1,0,0)	(v_0, v_1) (v_0, v_2)
2	$\underline{\mathsf{F}_1} \rightarrow \mathsf{F}_3 \rightarrow \mathsf{F}_2$	v_1:(1,1,0)	(v_1, v_3)
3	$\underline{\mathsf{F}_3} \rightarrow \mathsf{F}_2$	v_3:(1,1,1)	(v_3, v_4)
4	$\underline{\mathsf{F}_2} \rightarrow \mathsf{F}_4$	v_2:(1,0,0)	(v_2, v_4)
5	$\underline{\mathsf{F}_4}$	v_4:(1,1,1)	(v_4, v_5)
6	$\underline{\mathsf{F}_5}$	v_5:(1,1,1)	$\emptyset$

Figure 2: Classic vs. new reachability algorithm: Figures (c) and (d) depict the traversal traces of MS-BFS and MITra-RCH, respectively, via (i) $\mathbb{F}$, in which the "top" frontier F being explored in the round is underlined, (ii) vertices with their track in F (*e.g.,* v_4 : (1, 1, 0) means track($v_4, s_0/s_1$) = *True* and track(v_4, s_2) = *False*), and (iii) edges accessed when exploring F.

As shown in Fig. 1(b), after exploring u_1 in F_i in round j, $\mathcal{A}_M$ assigns v_h to (i) F_n for both sources s_1 and s_2 since rank(v_h, s_1) and rank(v_h, s_2) are in $[n \cdot \delta, (n+1) \cdot \delta)$, and (ii) F_0 for s_k since rank(v_h, s_k) $\in [0, \delta)$. When $\mathcal{A}_M$ fetches F_n and explores v_h of F_n in round m, the traversals and computations for s_1 and s_2 are then shared at v_h.

Example 1: BFS is commonly used for reachability computation on IP and road networks [18, 45]. It traverses graphs in rounds, starting from the source, and explores all vertices at the present depth prior to moving on to the next depth level. It can be acquired in $\mathcal{A}$ with specification in Table 1 for BFS (row 1), *i.e.,* by setting #round as rank, $\mathbb{F}$ to list and δ to 1. With the same frontier specification, multi-instance BFS, denoted by MITra-BFS, can be captured by $\mathcal{A}_M$ by populating vertex properties (*e.g.,* rank) of $\mathcal{A}$ to multiple sources.

As an example, consider G in Fig. 2(a) and three sources s_0, s_1 and s_2 (*i.e.,* v_0, v_1 and v_3) in Fig. 2(b). The trace of MITra-BFS for v_0, v_1 and v_3, in the language of $\mathcal{A}_M$, is depicted in Fig. 2(c).

Initially, $\mathbb{F}$ is empty and the ranks for all vertices and all sources are `undefined` except rank(v_0, s_0), rank(v_1, s_1) and rank(v_3, s_2) are set to `#round`, *i.e.,* 0. MITra-BFS creates frontier F_0 for v_0, v_1 and v_3 with frontier domain $[0, 1)$ covering their ranks. It sets track(v_0, s_0), track(v_1, s_1) and track(v_3, s_2) to *True* for F_0, and adds F_0 to $\mathbb{F}$.

In the first round, as shown in row 1 in Fig. 2(c), it pops out the "top" (only) frontier from $\mathbb{F}$, *i.e., F_0* for exploration: it first accesses edge (v_0, v_1) of v_0 with only track(v_0, s_0) = *True*; hence, it marks v_1 as discovered for source s_0 (*i.e.,* BFS edge function in Table 1), and updates rank(v_1, s_0) to `#round`, *i.e.,* 1 (step (2a) of $\mathcal{A}_M$); since rank(v_1, s_0) is changed, it *assigns v_1 for source s_0* to new frontier according to the updated rank(v_1, s_0), which is F_1 that does not exist in $\mathbb{F}$; hence, it creates frontier F_1 for v_1 and adds F_1 to $\mathbb{F}$ (step (2b) of $\mathcal{A}_M$). Similarly, it accesses (v_0, v_2) for source s_0, (v_1, v_3) for s_1 and (v_3, v_4) for s_2; $\mathcal{A}_M$ adds v_2, v_3 and v_4 to F_1 that already exists in $\mathbb{F}$. In round 2, as shown in row 2 of Fig. 2(c), $\mathbb{F}$ pops out the next "top" (only) frontier F_1 from $\mathbb{F}$ for exploration. $\mathcal{A}_M$ continues until no frontiers remain in $\mathbb{F}$ (row 4 in Fig. 2(c)).

In total, MITra-BFS accesses 10 edges for all three queries. It shares the access to edge (v_4, v_5) for both s_0 and s_1 in round 3 as both track(v_4, s_0) and track(v_4, s_1) are *True* in F_2. □

Properties. The frontier-ranking model has distinct properties.

(1) Abstraction. It normalizes traversal logic via vertex ranks, independent of computation logic cast in edge functions. This makes it possible to develop a framework approach to composing MIP algorithms, by abstracting multi-instance traversing away from users.

(2) Effectiveness. By "ranking" vertices to frontiers via numeric operations, it groups traversals from multiple sources according to vertex ranks, enabling shared cost of traversal and edge function computation across instances. By picking different vertex ranks, one can express existing sophisticated one-off MIP algorithms or even compose new MIP algorithms with provably higher sharing capabilities.

(3) Expressiveness. Frontier-ranking model is more expressive than existing graph computation models, even for single-instance algorithms. This allows us to develop MIP algorithms for a wider range of queries and algorithms than existing frameworks.

Below we formally prove and discuss properties (2) and (3) in Sections 3.2 and 3.3, respectively. We will delve into (1) in Section 4.

3.2 Effectiveness

We demonstrate the effectiveness of the frontier-ranking model for expressing existing one-off MIP algorithms and moreover, composing new MIP algorithms with provably higher degree of sharing, by simply playing with vertex ranks. We also show the link between vertex ranks and the capability of sharing across instances.

Expressing one-off algorithms. The frontier-ranking model can express state-of-the-art one-off MIP algorithms. Indeed, one can verify that MITra-BFS of Example 1, exactly captures MS-BFS [48], the state-of-the-art multi-instance BFS algorithm. Similar to MS-BFS, one can also cast the one-off MS-Dijkstra [57] as a frontier-ranking algorithm with vid as vertex ranks, δ = 1, $\mathbb{F}$ as a heap, and the typical edge relaxation operation of SSSP algorithms [16] as the edge function (row 4 of Table 1; see more in Section 4).

Deducing new algorithms. Frontier-ranking model abstracts traversal logic by means of vertex ranks, making it possible to provide an unified intuitive interface for us to compose new MIP algorithms by simply playing with vertex ranks. As an example, we show how to derive a new MIP algorithm for reachability queries from MS-BFS, with *provably* higher level of sharing than MS-BFS.

Multi-instance reachability. As discussed above, MS-BFS [48], the state-of-the-art MIP algorithm and the natural choice for multi-instance reachability queries [45], can be modeled in $\mathcal{A}_M$ as MITra-BFS. By simply changing its vertex ranks from `#round` to vid and retaining everything else (row 2 of Table 1), we derive a new algorithm from MS-BFS, denoted by MITra-RCH, that is even more effective than MS-BFS for reachability due to higher level of sharing.

Example 2: Continue with Example 1. We show that by simply changing vertex ranks from `#round` to vid, MITra-RCH is able to reduce the cost of MS-BFS for sources $s_0(v_0)$, $s_1(v_1)$, $s_2(v_3)$ over G in Fig. 2. The traversal trace of MITra-RCH is shown in Fig. 2(d).

More specifically, $\mathbb{F}$ is initially empty and the ranks for all vertices and all sources are `undefined` except that rank(v_0, s_0) is the vid of v_0, *i.e.,* 0, rank(v_1, s_1) is 1, and rank(v_3, s_2) is 3 (assume *w.l.o.g.* that the vid of v_i is i). It then creates three frontiers $\mathsf{F}_0 = \{v_0\}$, $\mathsf{F}_1 = \{v_1\}$ and $\mathsf{F}_3 = \{v_3\}$ such that the domain of $\mathsf{F}_i (i \in \{0, 1, 3\})$ covers the rank of v_i. It sets track(v_0, s_0), track(v_1, s_1) and track(v_3, s_2) to *True*. It then adds the three frontiers to $\mathbb{F}$ as a list $\mathsf{F}_0 \rightarrow \mathsf{F}_1 \rightarrow \mathsf{F}_3$.

In round 1, as shown in row 1 in Fig. 2 (d), MITra-RCH pops out the "top" frontier from $\mathbb{F}$, *i.e., F_0* for exploration. It first accesses edge (v_0, v_1) of v_0, mark v_1 as discovered for source s_0 (*i.e.,* BFS edge function), and updates the rank of v_1 for source s_0, *i.e.,* $\mathsf{rank}(v_1, s_0)$, from `undefined` to 1 since only $\mathsf{track}(v_0, s_0)$ is *True*. It then *assigns* v_1 *for source* s_0 to new frontier according to the updated $\mathsf{rank}(v_1, s_0)$, which is F_1 that already exists in $\mathbb{F}$; hence, it simply sets $\mathsf{track}(v_1, s_0)$ to *True* for F_1. Similarly, it traverses (v_0, v_2), *assigns* v_2 *for source* s_0 to a new frontier F_2, which does not exist in $\mathbb{F}$. So it creates F_2 for v_2 and appends F_2 to $\mathbb{F}$, which is now $\mathsf{F}_1 \rightarrow \mathsf{F}_3 \rightarrow \mathsf{F}_2$.

Along the same lines, it pops and explores F_1 from $\mathbb{F}$ in round 2 (row 2 in Fig. 2 (d)). This continues until $\mathbb{F}$ becomes empty (row 6).

In total, MITra-RCH accesses 6 edges, in contrast to 10 by MS-BFS. Indeed, MITra-RCH not only exploits sharing across sources for vertices at the same depth level of MS-BFS, it also "aligns" and shares computations across different depth levels of MS-BFS. This gives MITra-RCH more "shared accesses" to edges for the sources, as reflected by the number of $1's$ for vertices in the frontiers of MITra-RCH, *i.e.,* the $\underline{\mathsf{F}}$ column of Fig. 2(d). For instance, (v_3, v_4) is accessed only once for all three sources by MITra-RCH in round 3 as the track of v_3 for all three sources are *True* for F_3, while it is accessed three times by MS-BFS, one for source s_2 in round 1, one for source s_1 in round 2, and one for source s_0 in round 3 (Fig. 2(c)). □

The higher level of sharing of MITra-RCH over MS-BFS is not a coincidence: MITra-RCH enables better sharing for *each and every* input. Denote by $\mathsf{cost}_A(G, S)$ the number edge accesses when answering all sources in S over G by algorithm A. Then we have:

Theorem 1: *For* every *graph* G *and* every *set* S *of source vertices for reachability,* $\mathsf{cost}_{\mathsf{MITra\text{-}RCH}}(G, S) \leq \mathsf{cost}_{\mathsf{MS-BFS}}(G, S)$. □

From ranking to sharing. Theorem 1 is not a coincidence. Indeed, it is the implication of a connection between vertex ranks and the level of sharing of multi-instance frontier-ranking algorithms.

In general, vertex ranks that are more "permissive" give us more sharing. Consider two vertex ranks, namely, $\mathsf{rank}_1()$ and $\mathsf{rank}_2()$. We say that rank_1 is more *sharing-permissive* if for any vertex u of G, any sources s and s', the probability of $\mathsf{rank}_1(u, s) = \mathsf{rank}_1(u, s')$ is no lower than that of $\mathsf{rank}_2(u, s) = \mathsf{rank}_2(u, s')$. That is, rank_1 is more permissive than rank_2 if vertices are more likely to be ranked into the same frontier for the same source with rank_1.

With the same edge function and frontier specification, *i.e.,* $\mathbb{F}$ and δ, a frontier-ranking algorithms with vertex ranks that are more sharing-permissive will have a higher degree of sharing across instances. This has the following immediate implications.

(1) vid > #round. Algorithms with vid as ranks allow more sharing than those with #round as $\mathsf{rank}(u, s) = \mathsf{rank}(u, s')$ in all cases, *i.e.,* vid permits the highest level of sharing among all ranks. This also justifies the "instance-better" result of MITra-RCH in Theorem 1 .

(2) #round $\gtrapprox$ ans. Algorithms with #round as ranks usually have better sharing than those use ans, since vertices u visited by sources s and s' in the same round will be assigned with the same #round, *i.e.,* $\mathsf{rank}(u, s) = \mathsf{rank}(u, s')$, irrelevant of edge functions. In contrast, with ans the ranks of u for s and s' depend on edge functions and may not converge although they both visit u in the same round.

(3) $\mathbb{F}$ *and* δ. Frontier parameters $\mathbb{F}$ and δ are also relevant. Indeed, if the domain of ans is small and δ is large, ans can have more sharing than #round since u may be ranked to the same frontier for s and s' by ans even it is visited by them in different rounds.

3.3 Expressive Power

We next formally examine the expressiveness of frontier-ranking algorithms, by comparing with computation models of popular general-purpose graph frameworks. Since these frameworks and models cannot express multi-instance algorithms, we focus on single-instance graph traversal algorithms only, in favor of these models.

Models. We consider two classic models: *vertex-centric* underlying distributed graph frameworks, *e.g.,* Pregel [32] and *edge-traversal* model adopted by in-memory parallel frameworks, *e.g.,* Ligra [45].

(a) Vertex-centric model. Underlying most distributed graph frameworks [1, 19, 23, 30, 32, 49, 55] is the vertex-centric model. A vertex-centric algorithm runs in supersteps. In each superstep, every vertex of G invokes the same user program called vertex function in parallel; the output of an invocation at a vertex v of graph G is a message that is sent to all neighbors of v, as the input of their next superstep.

(b) Edge-traversal model. The edge-traversal model serves as the abstraction for shared memory parallel graph frameworks [45, 47]. It traverses G by edges and uses an edge function for computation logic. Different from frontier-ranking algorithms, it adopts a BFS-like traversal logic: when exploring a frontier, only *a single* frontier is generated for the immediate next round. When exploring a vertex u, finding neighbors of u is parallelized in frameworks of the model.

Simulation Theorem. Following [50], we next compare the expressive power of all three models via *model simulation.* Specifically:

- We say that model $\mathcal{M}$ can *optimally simulate* model $\mathcal{M}'$ if there exists a compilation algorithm that transforms any algorithm P' of $\mathcal{M}'$ with cost C to algorithm P of $\mathcal{M}$ with cost $O(C)$.
- A model $\mathcal{M}$ can *practically simulate* model $\mathcal{M}'$ if there exists a compilation algorithm that transforms any algorithm P' of $\mathcal{M}'$ with cost C to algorithm P of $\mathcal{M}$ with cost $O(C) + O(\log |V|)$.

Note that, if $\mathcal{M}$ optimally simulates $\mathcal{M}'$, it also practically simulates $\mathcal{M}'$, but not vice versa. Intuitively, if $\mathcal{M}$ optimally simulates $\mathcal{M}'$ but $\mathcal{M}'$ cannot simulate $\mathcal{M}$, then $\mathcal{M}$ is strictly more expressive than $\mathcal{M}'$. We next specify the cost of algorithms for the models.

We measure the cost of algorithms of all three models as the total number of edge accesses. To favor the vertex-centric model, we assume that its communication is completely free and we only count the number of *active* edge accesses as the cost of a vertex-centric algorithm, where an access to edge (u, v) is active in superstep $k+1$ only if the output of the vertex function at u changes in superstep k.

The frontier-ranking model is more expressive than vertex-centric and edge-traversal, even for single-instance graph traversals only.

Theorem 2: *(1) Vertex-centric and edge-traversal single-instance algorithms can be* optimally *simulated by frontier-ranking algorithms.*

(2) There exists single-instance frontier-ranking algorithm that cannot be practically *simulated by vertex-centric or edge-traversal.* □

Theorem 2 verifies that frontier-ranking algorithms have more expressive power for single-instance algorithms. In addition to this, since frontier-ranking model can express one-off style MIP algorithms that are beyond the reach of vertex-centric and edge-traversal models, taken together we have that the frontier-ranking model has greater expressive power than the traditional models.

ALGORITHM 1: MITra-RCH/MITra-BFS (multi-instance Reachbility/BFS)

```
1 #define $\mathbb{F}$ list; #define $\delta$ 1; #define rank vid ; // BFS: #define rank #round
2 initialize vertex properties <bool> ans ; // ans[$v$][$s$]: whether $v$ is reachable from $s$
.......................................................................................
3 Function EdgeFunc($u$, $v$)                              /* for both Reachability and BFS */
4     ans[$v$] ← mitra_Or(track[$u$], ans[$u$], ans[$v$]);
```

ALGORITHM 2: MITra-BellF/MITra-Dijk/MITra-DS (multi-instance SSSP)

```
1 #define $\mathbb{F}$ list; #define $\delta$ 1; #define rank #round ; // frontier and vertex rank
  for Bellman-Ford; pick corresponding configurations for Dijkstra and $\Delta$-stepping from Table 1
2 initialize vertex properties <int> ans ;          // ans[$v$][$s$]: the distance from $s$ to $v$
.......................................................................................
3 Function EdgeFunc($u$, $v$)              /* The distance function ($f(e)$) in Table 1 */
4     temp ← mitra_Add(track[$u$], ans[$u$], $w(u, v)$);
5     ans[$v$] ← mitra_Min(track[$u$], ans[$v$], temp);
```

4 THE MITRA FRAMEWORK

We are now ready to present the MITra framework. We start with the programming model of MITra (Section 4.1). We then discuss its internal workflow (Section 4.2) and implementation (Section 4.3).

4.1 Programming with MITra

Programming model. The programming interface of MITra consists of a preamble that configures frontiers and an edge function.

Preamble. MITra allows users to declare (a) configurations for vertex ranks, frontier width δ and frontier structure $\mathbb{F}$, which together instruct MITra the traversal logic that user wants to perform; and (b) any number of runtime vertex properties for the edge function.

Among them, a mandatory vertex property is ans that records answers of each vertex, *e.g.,* distance to the sources for SSSP. Similar to rank, each property X of a vertex u, denoted by X[u], is an array of values such that X[u][i] is the X-property of u for source s_i.

Edge function. The main interface of MITra is EdgeFunc, which allows users to specify computation logic $f(e)$ on edges $e = (u, v)$ by instructing how vertex properties are propagated from u to v.

The key to EdgeFunc is to express MIP computations over $e = (u, v)$ for multiple source vertices that visit v from u in the same step. To assist this, MITra automatically maintains track, a structure that records those sources that are traversing e (Section 3.1), and uses it to apply computation logic of EdgeFunc to only those sources that are visiting v from u, with a *single* invocation to EdgeFunc on e.

As shown in Table 1, by specifying EdgeFunc, vertex ranks, $\mathbb{F}$ and δ, MITra can express various multi-instance graph computations.

MITra **programs**. We show how to write MIP programs via MITra.

Graph search. Using MITra, the user programs for multi-instance Reachability and BFS, *i.e.,* MITra-RCH and MITra-BFS as discussed in Section 3, are exactly the same except their vertex ranks.

More specifically, as shown in Algorithm 1, the preamble of the program specifies the configuration of frontier and vertex ranks (line 1), and declares property ans such that ans[v][i] = *True* indicates that v is reachable from source s_i (line 2).

Both algorithms use a one-line edge function (line 4) that naturally extends single-source BFS with `mitra_Or`, one of the dedicated operators from MITra for deriving edge functions from traditional single-source algorithms: specifically, for each source s_i, it updates ans[v][i] with ans[u][i] $\vee$ ans[v][i] *if* track[u][i] *is True.*

Shortest path. Similarly, multi-instance shortest path (SSSP) algorithms, *e.g.,* Bellman-Ford, Dijkstra and Δ-stepping, also share the same EdgeFunc when written with MITra, despite their different traversal logic. As shown in Algorithm 2, they only differ in the preamble for vertex ranks and frontier setup (recall rows 3-5 in Table 1).

Indeed, their edge function is almost the same as that of *e.g.,* a text-book single-instance SSSP algorithm (lines 4-5 of Algorithm 2), carrying out "edge relaxation" [16] that relaxes the tentative distances ans[v] of v by checking whether ans[v] can be reduced by ans[u]+$w(u, v)$. The only difference is that we replace arithmetic operations in the relaxation, namely, `Add` and `Min`, with their MITra versions `mitra_Add` and `mitra_Min`, respectively. Specifically, for each source s_i, `mitra_Add` updates temp[i] with ans[u][i] + $w(u, e)$ if track[u][i] = *True* (line 4); similarly for `mitra_Min` (line 5).

Features. The MITra framework has the following properties.

(1) Edge-centric programming. MITra advocates an edge-centric programming interface: users only need to specify the computation logic via edge function from the perspective of an edge, independent of the traversal logic that is handled by the preamble via declared vertex ranks and frontier specification. This makes MITra programs for algorithms with distinct traversal logics almost identical, *e.g.,* three SSSP algorithms, or BFS vs. Reachability as shown above.

(2) Ease-of-use. In addition, it provides the MITra variant of logical and arithmetic operators, *e.g.,* `mitra_Or` and `mitra_Add`, that help users re-use edge functions from classic single-instance algorithms and populate them for EdgeFunc of multi-instance computations by simply replacing such operations with their MITra versions.

(3) Model obliviousness. Users configure traversal logic by specifying vertex ranks and frontiers in a preamble, and are oblivious to the housekeeping of *e.g.,* computing vertex ranks, ranking vertices to frontiers and maintaining track for the MITra operators.

These make MITra a convenient tool for users to implement MIP algorithms. For instance, it takes about 20 lines of code for both MITra BFS and Dijkstra (full program), as opposed to 550 and 200 lines (core function only) for MS-BFS [48] and MS-Dijkstra [57], respectively, while they have comparable performance (see Section 6).

4.2 From MITra programs to Frontier-Ranking

We next show how MITra synthesizes and executes a multi-instance frontier-ranking algorithm from a MITra program, *e.g.,* Algorithm 1.

As shown in Algorithm 3, MITra automatically generates frontiers according to the preamble, and carries out multi-instance traversing by following the frontier-ranking model $\mathcal{A}_M$ (Section 3.1), without user interference or algorithm-dependent knowledge.

Specifically, MITra first registers frontier and vertex ranks declared in the preamble (lines 1-3), and initializes vertex properties (line 4); in particular, it sets the rank of all vertices to `undefined` except the sources of which the rank is determined by the rank definition. MITra initializes $\mathbb{F}$ from sources according to their ranks (line 5). It then iteratively explores frontiers in $\mathbb{F}$ via procedure MTraverse, by following step (2) of model $\mathcal{A}_M$ (lines 6-8). Finally, it collects and returns ans[u] for all sources from all vertices u (line 9).

MTraverse visits each neighbor v of each vertex u in the current frontier F (lines 11-15), applies EdgeFunc to (u, v) (line 14), and pops

ALGORITHM 3: Full MITra algorithm (with internal functions)

```
Input: Graph $G(V, E)$, sources $s_1, \ldots, s_k$.
Output: for each $v \in V$ and $i \in [1, k]$, the answer of $v$ for $s_i$.
1  #define $\mathbb{F}$ frontier_structure; $\delta$ frontier_width;
2  #define <type> rank; <type> ans;
3  #define<type> $X_1$; ...; <type> $X_p$;                // declare vertex properties
4  initialize vertex properties;
5  initialize $\mathbb{F}$ from $s_1, \ldots, s_k$;
6  F ← $\mathbb{F}$.pop() ;                                // frontier for round 1
7  while F ≠ nil do
8      F ← MTraverse(F, EdgeFunc);
9  return ans of all vertices;

10 Procedure MTraverse(F, EdgeFunc)
11     foreach $u \in$ F do
12         foreach edge $(u, v)$ of $u$ in $G$ do
13             pre ← ans[$v$];
14             EdgeFunc($u$, $v$);                         // invoke user input edge function
15             UpdateF($v$, pre);                          // assign v to future frontiers
16     return $\mathbb{F}$.pop();        // pop a frontier for the next round; nil if $\mathbb{F}$ is empty

17 Procedure UpdateF($v$, pre_ans)
18     $\omega$ ← mitra_Neq(pre_ans, ans[$v$]);
19     oFids ← mitra_Div($\omega$, rank[$v$], $\delta$);
20     updaterank($\omega$, rank[$v$]) ;                  // built-in rank updates in MITra library
21     nFids ← mitra_Div($\omega$, rank[$v$], $\delta$);
22     foreach $i \in [1, k]$ and $\omega[i] = 1$ do
23         $\mathbb{F}$[oFids[$i$]].track[$v$][$i$] ← 0;
24         if $\mathbb{F}$[nFids[$i$]] = nil then // $\mathbb{F}$ has no frontier F with F.index = nFids[$i$]
25             create a new frontier $F_n$; $F_n$.index ← nFids[$i$];
26             $\mathbb{F}$.addFrontier($F_n$) ;   // $F_n$ can now be referenced as $\mathbb{F}$[nFids[$i$]]
27         $\mathbb{F}$[nFids[$i$]].track[$v$][$i$] ← 1;
```

out a new frontier from $\mathbb{F}$ for the next iteration (line 16). The crux is to (1) update rank[v] and (2) track progress of all the sources.

(1) Specifically, it updates rank[v][i] after visiting (u, v) only if EdgeFunc changes ans[v][i]; this avoids unnecessarily putting v to future frontiers for s_i. It does this by comparing the values of ans[v] before and after the invocation of EdgeFunc, via MITra operator **`mitra_Neq`** (line 18), and updates rank[v][i] for sources s_i of which ans[v][i] are changed (line 20). It also decides frontiers to which v needs to be assigned for s_i, via **`mitra_Div`** (line 21).

(2) It keeps track of traversal by maintaining track[v][i] for each source s_i when v is assigned to frontiers in $\mathbb{F}$ for future iterations (lines 22-27). When it assigns v to a new frontier F_n for s_i, it first removes v from the existing frontier for s_i by setting its track[v][i] to 0 (line 23). It then adds v to F_n by setting track[v][i] to 1 for F_n (lines 27); if such F_n is not yet in $\mathbb{F}$, it creates one first (lines 25-26).

Example 3: Recall the MITra program for Reachability in Algorithm 1. The full algorithm that MITra synthesizes from Algorithm 1 is exactly Algorithm 3 "instantiated" with (a) lines 1-4 adopted from the preamble of Algorithm 1 and (b) EdgeFunc (line 14) from that of Algorithm 1. One can readily verify that its trace over G of Fig. 2(a) for sources in Fig. 2(b) is exactly the MITra-RCH trace of Fig. 2(d) (recall Example 2). Similarly, the synthesized algorithm from Algorithm 1 with #round as ranks is exactly MITra-BFS (MS-BFS) and would give us the trace in Fig. 2(c). Along the same lines, one can get full frontier-ranking algorithms for Dijkstra, BellmanFord and Δ-stepping by instantiating Algorithm 3 with Algorithm 2. □

4.3 Implementation of MITra

Overview. MITra consists of two major components:

(1) User interface provides (a) a configuration module that allows users to specify frontier parameters and (b) an application module with which one can declare vertex properties and plug in an edge function to express the application-specific computation logic.

(2) MITra *library*. At its core is the MITra library, which assists users to adopt edge functions from classic single-instance algorithms and synthesize multi-instance frontier-ranking algorithms.

Implementation. We next discuss key details of the MITra library.

Frontiers. For sparse graphs, a frontier F is implemented as a map, where keys are vertices v in F and the mapped value of v is track[v] for F, encoded as a `bit` array of size k (the number of sources). For dense graphs, MITra uses arrays to represent frontiers for higher access efficiency. Specifically, F is a 2-dimensional `bit` array of size $|V| \times k$ such that F[v][i] is *True* if $v \in$ F and track[v][i] is *True*.

Unlike track that is local to frontiers, all vertex properties, *e.g.*, `rank` and `ans`, are global and hence implemented as arrays.

MITra *operators*. The library provides MITra variants of common logical and arithmetic operators, *e.g.*, **`mitra_Or`** in Algorithms 1. A MITra operator **`mitra_OP`**(track[u], X[u], X′[v]) selectively applies operator **OP** (*e.g.*, **`Add`**) to vertex properties X[u] and X′[v] only for sources s_i if track[u][i] = *True*. MITra operators allow users to simplify their MITra programs by conveniently adopting EdgeFunc from traditional single-instance algorithms (*e.g.*, Algorithm 1 and 2).

Extension. MITra implements two optional functions that give even more flexibility in composing MIP algorithms. (1) One may optionally specify a *vertex function* $f_v(u)$ that enables MITra to express traversals that, *e.g.*, only explore a frontier vertex u when certain conditions are met. (2) MITra also allows postround, a *post-hoc function* that operates at the end of each round. This can be helpful when the computation logic (*e.g.*, PageRank) requires to aggregate over properties of vertices visited in a round before moving on to the next.

Optimization. The design of MITra allows effective optimizations.

SIMD. MITra operators are optimized with SIMD, via *e.g.*, Intel intrinsics [2] that support vectorized mask operations, by treating track[u] as mask, *e.g.*, **`mitra_Add`**(track[u], ans[u], $w(u, v)$) is implemented as `_mm256_mask_add_epi32(inf, track[u], ans[u], _mm256_set1_epi32(w(u, v)))`. This ensures that shared traversal (EdgeFunc invocation) indeed leads to shared computation.

Tracking-free traversal. Recall that MITra tracks the traversal progress of all sources via the structure track, and uses it in MITra operators to update vertex properties only for the correct set of sources.

However, there exist algorithms for which such explicit progress tracking is not necessary. For them, MITra implements *tracking-free traversal* (TrackFree), an optimization that enables us to bypass references to track and propagate vertex properties of u to v for all sources. Specifically, this is realized in MITra via the MITra operators, which, if TrackFree is enabled, are implemented via standard SIMD operation without mask, *e.g.*, **`mitra_Add`**(track[u], ans[u], $w(u, v)$) would simply be `_mm256_add_epi32(ans[u], _mm256_set1_epi32(w(u, v)))`. This allows sources that are not visiting u to take "free rides" of those are, without extra cost.

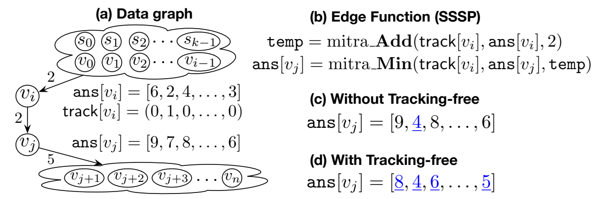

(a) Data graph
s_0 s_1 s_2 $\cdots$ s_{k-1}
v_0 v_1 v_2 $\cdots$ v_{i-1}
2
v_i
2
v_j
5
v_{j+1} v_{j+2} v_{j+3} $\cdots$ v_n
$\texttt{ans}[v_i] = [6, 2, 4, \ldots, 3]$
$\texttt{track}[v_i] = (0, 1, 0, \ldots, 0)$
$\texttt{ans}[v_j] = [9, 7, 8, \ldots, 6]$
(b) Edge Function (SSSP)
$\texttt{temp} = \text{mitra\_}\textbf{Add}(\texttt{track}[v_i], \texttt{ans}[v_i], 2)$
$\texttt{ans}[v_j] = \text{mitra\_}\textbf{Min}(\texttt{track}[v_i], \texttt{ans}[v_j], \texttt{temp})$
(c) Without Tracking-free
$\texttt{ans}[v_j] = [9, \underline{4}, 8, \ldots, 6]$
(d) With Tracking-free
$\texttt{ans}[v_j] = [\underline{8}, \underline{4}, \underline{6}, \ldots, \underline{5}]$

Figure 3: TrackFree **Optimization**

ALGORITHM 4: MITra-SpMV (multi-instance SpMV with MITra)

```
1 #define 𝔽 list; #define δ 1; #define rank #round;
2 initialize properties <int> $\texttt{ans}_{\text{cur}}$, $\texttt{ans}_{\text{next}}$;
.................................................................
  Function EdgeFunc($u$, $v$)                /* The SpMV function ($f(e)$) in Table 1 */
3     temp ← mitra_Mul(track[$u$], $\texttt{ans}_{\text{cur}}[u]$, $w(u, v)$);
4     $\texttt{ans}_{\text{next}}[v]$ ← mitra_Add(track[$u$], $\texttt{ans}_{\text{next}}[v]$, temp)
5 Function postround
6     foreach $v \in V$ do $\texttt{ans}_{\text{cur}}[v]$ ← mitra_Set($\texttt{ans}_{\text{cur}}[v]$, 0) ;
7     swap($\texttt{ans}_{\text{cur}}$, $\texttt{ans}_{\text{next}}$);
```

Example 4: Consider a weighted graph G in Fig. 3(a), which has (1) a connected component C_1 with vertices $s_0, s_1, \ldots, s_{k-1}, v_0, v_1, \ldots, v_{i-1}$, (2) connected component C_2 with vertices $v_{j+1}, v_{j+2}, \ldots, v_n$, and (3) two vertices v_i and v_j with adjacent edges connecting C_1 and C_2; edge weights are also shown in Fig. 3(a). Consider SSSP query with k source vertices $S = \{s_0, s_1, \ldots, s_{k-1}\}$ from C_1. Recall from Section 4.1 that the ans vertex property stores the tentative distance of each vertex in G from each source in S.

Assume in round r, a frontier F that contains vertex v_i is explored. The track$[v_i]$ of v_i in F is shown in Fig. 3(a), indicating that only source s_1 is visiting v_i for the current round, *i.e.,* track$[v_i][1]$=*True*. Upon visiting edge (v_i, v_j), the edge function for SSSP (*i.e.,* Fig. 3 (b)) is invoked for updating ans$[v_j]$ using ans$[v_i]$ via MITra operations masked by track$[v_i]$. One can verify that (1) without TrackFree, ans$[v_j]$ is updated to Fig. 3(c), *i.e.,* only ans$[v_j][1]$ is progressed from 7 to 4 (blue, underlined); In contrast, (2) with TrackFree, all k values of ans$[v_j]$ are updated as shown in Fig. 3(d). The immediate effect of TrackFree is that changes of all k values of ans$[v_j]$ are propagated earlier to component C_2, leading to faster termination. □

The benefit is two-fold. (a) Bypassing track eliminates tolls for maintaining track, yielding reduced overhead. (b) As shown in Example 4, by progressing earlier via free riding on MITra operations, one gets earlier termination and reduced EdgeFunc invocations.

Applying TrackFree changes the traversal logic, and hence may not apply to all queries. The rule of thumb is that if ans$[v][i]$ changes monotonically for each v and source s_i and converges to the same terminating value, independent of the order of edges on which EdgeFunc fires, then TrackFree applies to the query and all algorithms for the query. For instance, Reachability and SSSP queries have the property and benefit from TrackFree, while BFS cannot.

5 APPLICATIONS

We have seen how to program with MITra for multi-instance graph search and shortest path queries (Section 4). Below we further demonstrate MITra with graph analytical computations.

Sparse matrix-vector multiplication (SpMV). SpMV is a widely used kernel in graph analytics, *e.g.,* Graph/Recurrent Neural Networks [17, 40]. Generalized SpMV iteratively computes $x^{t+1} = Ax^t = \bigoplus_{i=1}^{n} A_i \otimes x^t$, where A is the adjacency matrix (transposed) of a graph G and A_i is the i-th row of A; x^t is a vector of values, each for a vertex of G; t is the number of iterations that have been completed; $\oplus$ and $\otimes$ are algorithm-specific semiring operators. To reduce notation, we simply adopt the standard + and × for $\oplus$ and $\otimes$, respectively, which yields the standard matrix-vector multiplication.

It has been a common practice to use graph computation frameworks for SpMV, by framing it as a graph traversal problem [17, 27, 42, 47]: if we view x as a dummy vertex v_x that is connected to vertices for which the corresponding entries in x is "non-zero", then SpMV reduces to a traversal of G starting from v_x. Naturally, multi-instance SpMV carries out k such SpMV iterations $Ax_1, \ldots, Ax_k$.

We next show how MITra helps with multi-instance SpMV, by presenting MITra-SpMV (Algorithm 4). In the preamble, it declares vertex ranks and configures frontiers according to Table 1 (line 1). It also registers $\texttt{ans}_{\text{cur}}$ and $\texttt{ans}_{\text{next}}$, where $\texttt{ans}_{\text{cur}}$ is the value of x from the last round and $\texttt{ans}_{\text{nest}}$ is its value updated by EdgeFunc in the current round. Initially, $\texttt{ans}_{\text{cur}}$ is set according to $x_j (j \in [1, k])$ and $\texttt{ans}_{\text{next}}$ is 0 for all vertices and all instances (line 2).

The edge function (lines 3-4) is the standard addition and multiplication implementation of $\oplus$ and $\otimes$, in MITra versions. Upon visiting v through edge (u, v), it first multiplies $\texttt{ans}_{\text{cur}}[u]$ by edge weight $w(u, v)$ and stores the value in array temp (line 3), and then adds temp to $\texttt{ans}_{\text{next}}[v]$ (line 4). In addition, MITra also specifies the post-hoc function postround (recall Section 4.3) to reset $\texttt{ans}_{\text{cur}}$ (lines 6) and switch $\texttt{ans}_{\text{cur}}$ and $\texttt{ans}_{\text{next}}$ for the next round (line 7).

PageRank. Given G and source vertex s, the Personalized PageRank (PPR) value of a vertex v for s, denoted by P_v, is the probability that a random walk in G from s terminates at v.

PPR algorithms are based on a recurrence equation [21, 36, 38]:

$$P^{t+1} = \alpha \mathbf{X} P^t + (1 - \alpha) * e_s,$$

where α is the damping factor, $\mathbf{X} = A\mathbf{D}^{-1}$, where $\mathbf{D}$ is the out-degree matrix of G (a diagonal matrix with the number of out-degree of each vertex), P^t is the PPR values at iteration t, and e_s is an identity vector ($e_s[v]$ is 1 when v is s and 0 otherwise). Intuitively, the first term is an SpMV and the second term is a personalization factor. Hence, PPR is often solved by adopting an SpMV algorithm with an addition of the personalization factor in each iteration [17, 38].

Multi-instance PPR computes for each vertex v in G, given a set S of source vertices, the PPR value of v for each $s \in S$. It can naturally be handled by adopting MITra-SpMV; we denote it by MITra-PPR.

Remark. Implementing algorithms with MITra does not incur extra tolls in terms of computation complexity. The overhead of MITra only comes from operations over rank and track, which take $O(1)$ time upon each visit to a vertex by a source. Hence, with only one source, the MITra algorithms we presented so far have the same complexity as their traditional counterparts. When there are k sources, in the worst case, the sources can be mutually independent and no computation sharing or complexity reduction is possible.

However, the actual cost of the algorithms varies significantly *w.r.t.* degree of sharing. Indeed, when an access to an edge e is shared by two sources s and s', their *traversal cost, e.g.,* fetching vertices and edge lists and maintaining frontiers, is completely shared. In

Table 2: Real-life data graphs and synthetic graphs

	Graphs	#vertices $\|V\|$	#edges $\|E\|$	degree	diameter [5]
dense	`Pokec` [4]	1,632,803	30,622,564	18.75	11
	`LiveJournal` [3]	4,847,571	68,993,773	14.23	16
	`Twitter` [8]	41,652,230	1,468,365,182	35.25	23
	`UKDomain` [9]	105,153,952	3,301,876,564	31.40	112
	`rMat` [15]	2^{23}-2^{27}	2^{27}-2^{31}	16	8-9
sparse	`UKTraffic` [10]	7,733,822	17,687,718	2.29	4604
	`DETraffic` [10]	11,548,845	26,872,465	2.33	3130
	`USTraffic` [11]	23,947,347	58,333,344	2.44	8315
	`EUTraffic` [10]	50,912,018	108,109,320	2.12	14427
	`Grid-2d` [45]	2^{23}-2^{27}	2^{25}-2^{29}	4	2896-11584

theory, the *computation cost*, *i.e.,* total work of EdgeFunc over e for s and s', cannot be reduced; however, in practice it still benefits from the shared access via *e.g.,* SIMD and reduced invocation cost.

6 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we evaluated the performance of MITra for common multi-instance graph computations.

6.1 Experimental Setup

Graphs. We used 8 real-life graphs and two graph generators, `rMat` [15] and `Grid-2d` [45], that respectively generate dense and sparse graphs of varying sizes (Table 2). Following[31, 45], we generated edge weights between $[1, \log|V|)$ uniformly at random.

Queries. Following [48, 57] that use vertices that are close to each other in the data graph as queries, we randomly sampled a seed vertex and run a BFS starting from the seed vertex to get k vertices as queries in data graphs. We generated 5 sets of queries for each data graph by varying k from 16, 32, 64, 128 to 256. We used the same source vertices as input for all compared methods in a test. We sampled three such seed vertices and generated three groups of queries for each data graph; the average is reported.

Implementation. We implemented MITra in C++. All the MITra arithmetic operators (Section 4.3) are available in both standard implementation that loops over sources and the version that employs Intel SIMD intrinsics (by default we used the later). As built-in library, we also implemented MITra-BFS, MITra-RCH, MITra-BellF, MITra-Dijk, MITra-DS, MITra-PPR and MITra-SpMV by using configurations in Table 1 and edge functions in Section 4.1 and 5.

Comparisons. We compared the performance of MITra algorithms with the following MIP methods (see Table 3 for a full list).

(a) Serial algorithms. We compared with serial algorithms for all 7 computations of Section 2. Among them, serial BFS/Reachability, Bellman-Ford and PPR use their built-in implementations in Ligra [45] from [6], a popular open-source graph framework. We implemented SpMV in Ligra following PPR. Since we are not aware of any framework that supports Dijkstra, we implemented a serial Dijkstra based on the classic implementation [16]; similarly for Δ-stepping [34].

(b) One-off algorithms. We also compared with MS-BFS [48] and MS-Dijkstra [57], the only two one-off algorithms we are aware of.

Configuration. The experiments were run on an r6i.8xlarge AWS EC2 instance, with 256 GB of memory. We used one thread by disabling hyperthreading for all the tests. Ligra uses `parallel_for` loops to explore vertices and iterate over their neighbors in parallel, where `parallel_for` is implemented by using OpenMP/Cilk [6, 45]. To avoid parallelization overhead over a single thread, we replaced such `parallel_for` with traditional `for` loops. Each experiment was run 3 times; the average is reported.

Table 3: MIP methods used in the experiments

Category	Computation	Method	Implementation
Serial	BFS/Reachability	Ligra-BFS	open source [6]
	Bellman-Ford	Ligra-BellF	
	PPR	Ligra-PPR	
	SpMV	Ligra-SpMV	
	Dijkstra	kDijkstra	implemented following [16]
	Δ-stepping	k´-stepping	implemented following [34]
One-off	BFS/Reachability	MS-BFS	open source [7]
	Dijkstra	MS-Dijkstra	implemented following [57]
MITra	BFS	MITra-BFS	this work
	Reachability	MITra-RCH	
	Bellman-Ford	MITra-BellF	
	Dijkstra	MITra-Dijk	
	Δ-stepping	MITra-DS	
	PPR	MITra-PPR	
	SpMV	MITra-SpMV	

6.2 Performance

Overall performance. Varying the number k of sources from 16 to 256, we tested the run time of all methods over all real-life graphs.

MITra Vs. *serial algorithms*. We first compared MITra with all serial algorithms in Table 3. The results are shown in Figures 4a-4i.

(1) MITra is efficient. Over all 8 graphs, MITra-BFS, MITra-RCH, MITra-BellF, MITra-Dijk, MITra-DS, MITra-PPR and MITra-SpMV is on average 8.73x, 33.45x, 10.38x, 19.7x, 2.76x, 7.13x and 7.71x faster than Ligra-BFS (BFS), Ligra-BFS (Reachability), Ligra-BellF, kDijkstra, k´-stepping, Ligra-PPR and Ligra-SpMV, respectively, up to 35.47x, 140.61x, 49.3x, 62.67x, 8.7x, 11.9x and 20.31x.

(2) The speedup of MITra over serial methods increases when more queries are used. For instance, MITra-RCH is 19.02x faster than Ligra-BFS over `LiveJournal` with 16 queries, while this goes up to 86.62x with 256 queries; similarly for other algorithms and graphs.

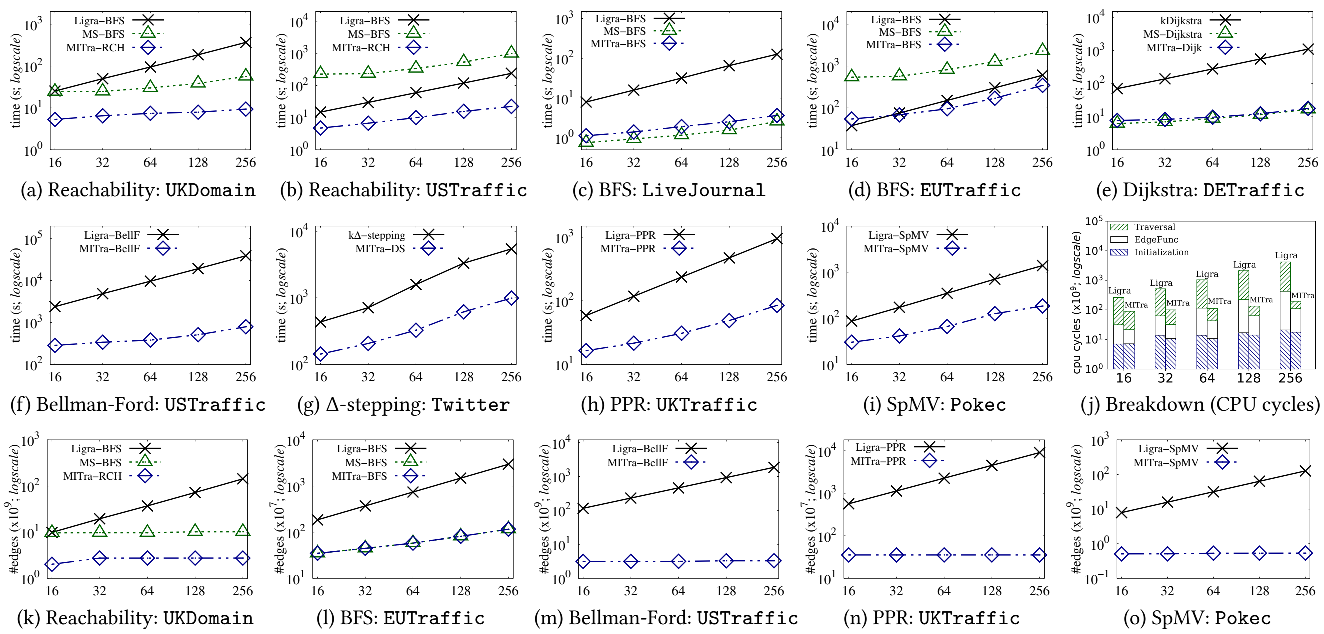
(3) MITra consistently outperforms serial algorithms for Reachability, Bellman-Ford, Dijkstra, PPR and SpMV in each and every test; it is consistently faster than Ligra-BFS over dense graphs, *e.g.,* `LiveJournal`, `Pokec`, `Twitter` and `UKDomain`.

However, over sparse graphs, *e.g.,* `EUTraffic`, Ligra-BFS is even faster than the one-off MS-BFS, and faster than MITra-BFS, when the number of queries is low, *e.g.,* 16 (Fig. 4d). This is because the cost of BFS edge functions is rather small and over sparse graphs the chance of sharing traversal across queries is lower than over dense graphs. Hence, the benefit of MIP could be cancelled out by the overhead when there is no sufficient sharing available.

MITra Vs. *one-off algorithms*. We compare MITra with one-off algorithms, *i.e.,* MS-BFS for BFS and Reachability, MS-Dijkstra for SSSP.

(1) For BFS, MITra has performance comparable to the heavily engineered MS-BFS for BFS queries, despite that it is a MIP framework and cannot implement algorithm specific optimizations in MS-BFS.

Indeed, MITra-BFS is even 4.49x faster than MS-BFS over all 4 sparse graphs in Table 2 while MS-BFS is 1.65x faster than MITra-BFS over the dense graphs. This is because MS-BFS implements specific optimizations for dense graphs, *e.g.,* aggregated neighbor



(a) Reachability: `UKDomain` (b) Reachability: `USTraffic` (c) BFS: `LiveJournal` (d) BFS: `EUTraffic` (e) Dijkstra: `DETraffic`

(f) Bellman-Ford: `USTraffic` (g) Δ-stepping: `Twitter` (h) PPR: `UKTraffic` (i) SpMV: `Pokec` (j) Breakdown (CPU cycles)

(k) Reachability: `UKDomain` (l) BFS: `EUTraffic` (m) Bellman-Ford: `USTraffic` (n) PPR: `UKTraffic` (o) SpMV: `Pokec`

Figure 4: Running time (Section 6.2) and `#edges` (Section 6.3) of all methods with varying number (k) of queries

processing, that add around 1.5x to 2x of speedup to MS-BFS according to [48]; they are not possible to be incorporated to MITra as a generic framework. Due to the frontier representation in MITra (Section 4.3), MITra-BFS is faster than MS-BFS over sparse graphs.

The case for MITra-Dijk vs. one-off MS-Dijkstra is similar. On average MS-Dijkstra is only 3.35% faster than MITra-Dijk.

(2) For Reachability, MITra-RCH is even 19.36x faster than MS-BFS on average (2.31x over dense graphs and 36.41x over sparse graphs). This is because, for Reachability, multi-instance BFS (even MS-BFS) is not optimal anymore due to the sharing opportunities that are missed by BFS. This also demonstrates the benefit of frontier-ranking model for systematically modeling and capturing sharing.

Breakdown. We further examined the efficiency of MITra by profiling its CPU consumption. We divide each algorithm into three parts: (a) initialization that prepares vertex properties and associated data structures, (b) computation logic, *i.e.,* edge function, and (c) traversal logic, which includes all other steps except those in (a) and (b). Using `perf`, we examined the CPU cycles consumed by each part with varying number of queries. The results of Ligra-BellF and MITra-BellF over `UKTraffic` are shown in Fig. 4j.

(1) The total number of CPU cycles used by MITra-BellF is smaller than Ligra-BellF in all cases, consistent with their running time.

(2) The degree of traversal sharing by MITra grows substantially with increasing number of queries. For instance, the traversal logic of Ligra-BellF takes 2.39x more CPU cycles than that of MITra-BellF with 16 queries, while this goes up to 42.89x with 256 queries.

(3) The reduction of MITra-BellF over Ligra-BellF on computation logic also grows with increasing number of queries, *e.g.,*Ligra-BellF costs 1.71x more CPU cycles on edge functions than MITra-BellF with 16 queries, while this goes up to 4.42x with 256 queries. The reduction is relatively lower than traversal cost in (2) above since merging execution of edge functions does not necessarily reduce the total amount of work; nonetheless, it still reduces CPU consumption due to reduced invocation cost, better data locality and SIMD.

6.3 Effectiveness of Sharing

To understand how MITra gains its speedups, we further evaluated the effectiveness of sharing across instances, by examining the total number of edge accesses (`#edges`) of all methods over all graphs.

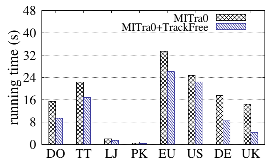
(1) As shown in Figures 4k-4o, MITra consistently accesses fewer edges (edge function invocations) than serial algorithms do. On average, serial BFS, Reachability, Bellman-Ford, Dijkstra, Δ-stepping and PPR and SpMV access 23.98, 41.62, 127.74, 64.44, 66.91, 96.75 and 102.46 times more edges than MITra-BFS, MITra-RCH, MITra-BellF, MITra-Dijk, MITra-DS, MITra-PPR and MITra-SpMV do.

(2) It is justified to use `#edges` as a conceptual level measure of MIP algorithms as `#edges` of all methods is largely consistent with their performance in Section 6.2. However, improvement over `#edges` does not always exactly match efficiency speedups. For instance, `#edges` of both one-off MS-BFS and MITra-BFS is lower than that of Ligra-BFS in all cases, while, however, Ligra-BFS is even faster over sparse graphs, *e.g.,* `EUTraffic`, for the reason discussed in Section 6.2. However, for all the other cases where edge functions are heavier than BFS or there is sufficient sharing, MITra and one-off algorithms still consistently outperform serial ones as expected.

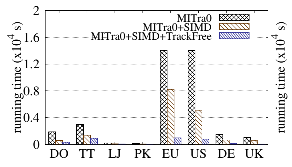
6.4 Effectiveness of Optimization

We also evaluated the effectiveness of optimization in MITra: SIMD and TrackFree (recall Section 4.3). For each algorithm we fixed the query number to 256 and varied the optimizations applied to it (if applicable) cumulatively and in the order: SIMD and TrackFree. Figure 5 reports the time of the algorithms.

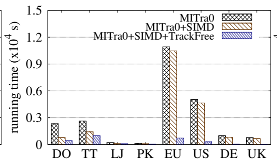
DO: `UKDomain` **TT**: `Twitter` **LJ**: `LiveJournal` **PK**: `Pokec` **EU**: `EUTraffic` **US**: `USTraffic`; **DE**: `DETraffic` **UK**: `UKTraffic` **MITra0**: MITra with both SIMD and TrackFree disabled.



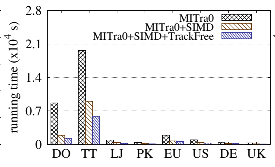
(a) Reachability



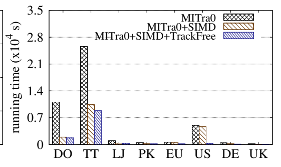
(b) Bellman-Ford



(c) Δ-stepping

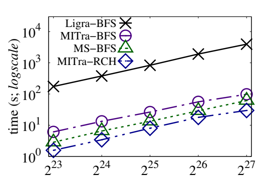


(d) PPR

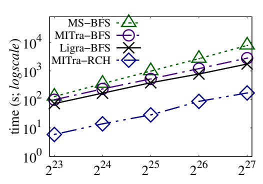


(e) SpMV

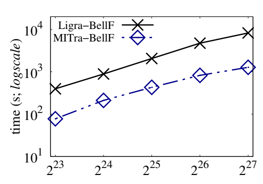
Figure 5: Effectiveness of SIMD and TrackFree optimization in MITra (Section 6.4)



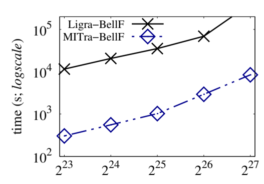
(a) `rMat`: BFS/Reachability



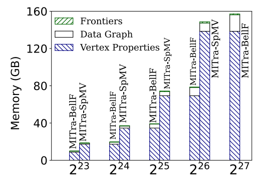
(b) `Grid-2d`: BFS/Reachability



(c) `rMat`: Bellman-Ford



(d) `Grid-2d`: Bellman-Ford



(e) `rMat`: Bell.-Ford/SpMV

Figure 6: Experimental results over synthetic graphs with varying number $|V|$ of vertices (Section 6.5)

SIMD. We find that SIMD provides an evident performance boost in MITra, *e.g.,* on average 1.96x, 1.42x, 1.61x, 2.27x and 2.2x of speedup for Bellman-Ford, Dijkstra, Δ-stepping, PPR and SpMV, respectively, over all graphs. MITra-BFS and MITra-RCH do not exploit SIMD as their edge function requires logical operations only that are already implemented with bitwise operators by default.

Tracking-free traversal (TrackFree**)**. TrackFree substantially improves MITra SSSP algorithms. It improves the efficiency of MITra with SIMD by 3.06x, 1.39x and 4.4x on average for Bellman-Ford, Dijkstra and Δ-stepping, respectively, for all queries over all graphs, reducing `#edges` by 2.69x, 1.18x and 3.23x.

TrackFree also demonstrates a noticeable speedup for Reachability, PPR and SpMV algorithms: it gives an average speedup of 1.16x, 1.48x and 1.34x for MITra-RCH, MITra-PPR and MITra-SpMV over MITra with SIMD respectively. Due to their traversal logic, TrackFree does not reduce their edge accesses; hence the improvement for them is not as good as for SSSP algorithms. Recall that TrackFree does not apply to BFS; hence we do not report BFS.

SIMD+TrackFree. Putting together, they improve plain MITra by 1.16x, 5.8x, 2.02x, 5.62x, 3.31x and 2.77x on average for Reachability, Bellman-Ford, Dijkstra, Δ-stepping, PPR and SpMV, respectively.

6.5 Scalability

Using `rMat` and `Grid-2d`, we evaluated the performance of MITra over dense and sparse synthetic graphs of varying sizes.

Scalability. We first tested the running time of all methods with 256 sources over dense and sparse graphs generated by `rMat` and `Grid-2d`, respectively. The results for BFS, Reachability and Bellman-Ford are shown in Figures 6a-6d.

(1) MITra-RCH is consistently the fastest among all BFS and Reachability methods over both dense and sparse graphs of all sizes, and its improvement over others even grows with larger graphs, *e.g.,*110.79x faster than Ligra-BFS over `rMat` with 2^{23} vertices, up to 132.9x with 2^{27} vertices (Fig. 6a); similarly for Bellman-Ford.

(2) For the same reason as in Section 6.2, Ligra-BFS is the best among all BFS methods over sparse `Grid-2d` graphs, but its speedup over MITra-BFS is moderate, *e.g.,* below 35% in all cases. MITra-BFS is consistently faster than MS-BFS, *e.g.,* by 1.29x over `Grid-2d` with 2^{23} vertices; the speedup increases to 2.79x with 2^{27} vertices.

Memory overhead. We examined the memory cost of all methods over graphs with varying sizes via `Intel VTune profiler`. The results for Bellman-Ford and SpMV over `rMat` graphs are shown in Fig. 6e; the results over `Grid-2d` are almost identical. We find that the memory footprint of MITra algorithms is dominated by the size of vertex properties, *e.g.,* over 92.5% for SpMV over `rMat` of all sizes. Moreover, the size is determined by property type and the numbers of vertices and sources. For instance, over `rMat` with 2^{27} (rMat27) vertices, the vertex properties of MITra-BellF take 138.5 GB of memory, which is the same as that of MITra-SpMV over rMat26 since MITra-SpMV uses twice as many as vertex properties (integers) as MITra-BellF (MITra-SpMV over rMat27 runs out of memory).

6.6 Discussion

Summary. We find that, despite its ease of use, on average MITra is 8.73x, 33.45x, 10.38x, 19.7x, 2.76x, 7.13x and 7.71x faster than serial algorithms for multi-instance BFS, Reachability, Bellman-Ford, Dijkstra, Δ-stepping, PPR and SpMV, respectively, and is comparable to one-off algorithm MS-BFS [48] for BFS and MS-Dijkstra [57] for SSSP. This is due to its capability of extracting sharing via reduced edge accesses, and effective optimizations underlying its interface.

Limitation. MITra has its limitations. (1) Algorithms with a lightweight edge function may not benefit from MITra over very sparse graphs, in particular with few sources, *e.g.,* BFS over `EUTraffic` with 16 sources. This is because with few sources over sparse graphs, the amount of traversal sharing is relatively low, which also leads to low computation sharing due to the lightweight edge functions. Hence the overhead of sharing may outweigh its benefits. However, with a heavier edge function, *e.g.,* SSSP algorithms, MITra is still effective over sparse graphs. (2) MITra does not help if the edge functions are heavy but mostly due to imperative instructions that cannot use MITra operators. For instance, regular path query (RPQ) algorithms [26] over graphs spend most time on examining the automata that encode the queries upon edge accesses, and hence may not be improved by MITra. (3) MITra does not apply to algorithms without input sources, *e.g.,* computing connected component [16].

Making practical use of MITra. Based on the observations, we give a general guideline for using MITra in practice. (a) We first decide whether the target algorithm **A** suits MITra, by referring to the discussion of limitations above. (b) If so, we then estimate the memory cost. Given memory budget M, data graph $G(V, E)$, a set S of sources and the number p of vertex properties of data type T used by the edge function, we calculate the number k of queries that could be processed in one go as $k = \frac{M-|G|}{p*|V|*sizeof(T)}$. One can divide the sources into $\lceil\frac{|S|}{k}\rceil$ groups for processing. (c) Finally, we decide the configuration and optimization for **A** cast in MITra, by referring to Section 3.2 for the choice of vertex ranks and Section 4.3 for optimization options.

7 RELATED WORK

We categorize related work as follows.

Multi-instance processing, often dubbed as multi-query processing, has been well studied for SQL [43, 44], XQuery [14], SPARQL [28, 39], subgraph [41] and regular path query (RPQ) [12]. They identify common sub-queries and re-use them to answer multiple queries. Instead, we consider graph traversals for the same queries but instantiated with multiple source vertices.

There has also been work on developing one-off MIP algorithms that are crafted and optimized for specific graph traversal problems. For example, MS-BFS [24, 48] is an MIP algorithm that executes BFS searches starting from multiple source vertices. It represents graphs as adjacency lists and develops implementation and heuristic techniques that share the access to the adjacent lists for multiple source vertices when possible. Similarly, multi-source algorithms have been developed for shortest path computations over multi-core machines [57] and distributed Hadoop clusters [29].

MITra differs from this as follows. (1) In contrast to one-off approaches that are algorithm specific, MITra is a generic framework for composing MIP algorithms for all common graph traversals. Conceptually, one-off algorithms, *e.g.,* MS-BFS, are an application of MITra. (2) In addition, MITra provides a unified abstraction of MIP algorithms, allowing us to deduce new algorithms and analyze their sharing capability. Indeed, by simply changing the vertex ranks of MITra-BFS, *i.e.,* MS-BFS framed in MITra, we obtained a new MIP algorithm that has provably higher level of sharing and is substantially faster than MS-BFS for Reachability. (3) One-off algorithms often target specific applications over certain type of graphs, *e.g.,* MS-BFS is engineered for small-world graphs. In contrast, as a framework MITra targets all types of graphs, sparse or dense. Indeed, the frontier implementation of MITra makes MITra-BFS even faster than MS-BFS over sparse graphs.

Graph computation frameworks. There has been a large number of general-purpose distributed [1, 19, 23, 30, 32, 49, 55] and parallel [20, 27, 45] graph frameworks. They target traditional single-instance graph algorithms and do not help with multi-instance computations. In addition, with the frontier-ranking model, MITra can express algorithms (*e.g.,* Dijkstra) beyond the expressive power of these frameworks, even for single-instance algorithms. MITra is orthogonal to and complements prior art in parallel frameworks. Indeed, one can incorporate the parallelism extraction methods of parallel frameworks into MITra by exploring vertices in each frontier in parallel, supporting parallel multi-instance computation.

Also related is the study of parallel frameworks for concurrent iterative graph computation jobs [31, 33, 37, 53, 56, 58, 60]. They optimize groups of jobs via scheduling to increase parallelism [33, 56, 58] and cache hit rate [31], which is often jointly guided by hardware and job characteristics. They do not explore any computation sharing across jobs, except for SimGQ [53, 54], which allows computation sharing across BFS-like iterative jobs in parallel.

Our work differs from the research as follows. (1) simGQ is restricted to BFS-like computations, which are only one instantiation of MITra with `#round` as `rank`, $\mathbb{F}$ as a list and $\delta = 1$. (2) simGQ gives no abstraction of multi-instance traversals; hence, it can neither characterize the sharing degree of MIP algorithms that MITra is capable of, nor compose algorithms with traversal logic different from BFS, *e.g.,* MITra-RCH, MITra-Dijk and MITra-DS. (3) simGQ builds on top of Ligra and does not have components similar to MITra operators and optimizations, *e.g.,* SIMD and TrackFree. As a result, it extracts less traversal sharing than MITra and cannot convert traversal sharing to computation sharing. Hence, even for algorithms that simGQ supports, *e.g.,* BFS and Bellman-Ford, its speedup over Ligra as reported in [53, 54] is much lower than MITra, *e.g.,* 5.63x (256 sources) and 1.67x (64 sources) by simGQ for BFS and Bellman-Ford, respectively, over `LiveJournal` vs. 35.61x (256 sources) and 2.88x (64 sources) by MITra. (4) Due to the higher level of sharing, MITra is evidently more effective with growing number k of sources, while simGQ sees only marginal improvements (cf. [53, 54]). (5) Similar to MS-BFS, simGQ is designed for small-world graphs while MITra targets all types of graphs.

8 CONCLUSION

We have proposed MITra, a framework for composing MIP graph algorithms that are as efficient as highly optimized one-off algorithms while being as simple as traditional single-instance algorithms.

There is naturally more to be done. One direction is to extend MITra for synthesizing parallel MIP algorithms over multi-core machines. A simple idea is to adopt Ligra's approach to parallelization, by employing OpenMP/Cilk for iterating over frontiers via "`parallel_for`" [6, 45]. While this works for BFS-like algorithms targeted by Ligra that produce one frontier at a time over shared-memory machines, it is much more challenging when it comes to generic frontier-ranking algorithms supported by MITra, which are more expressive and include *e.g.,* Dijkstra and Depth-First-Search that are known hard to parallelize. Another direction is to synthesize distributed MIP algorithms for vertex-centric systems, to share both computation and communication cost among instances.

ACKNOWLEDGMENTS

Cao is supported by Royal Academy of Engineering Research Fellowship RF\201920\19\319 and Edinburgh-Huawei Joint Lab. Buneman is supported by the Edinburgh-Huawei Joint Lab.

REFERENCES

[1] Giraph. https://giraph.apache.org/.

[2] Intel® intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[3] LiveJournal. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[4] Pokec. http://snap.stanford.edu/data/soc-Pokec.html.

[5] Snap Library. https://snap.stanford.edu/snap/description.html.

[6] The source code for Ligra framework. https://github.com/jshun/ligra.

[7] The source code for MS-BFS algorithm. https://github.com/mtodat/ms-bfs.

[8] Twitter. http://konect.cc/networks/twitter/.

[9] UK domain. http://konect.cc/networks/dimacs10-uk-2007-05/.

[10] UK/DE/EU Traffic. https://www.cc.gatech.edu/dimacs10/archive/streets.shtml.

[11] US Traffic. http://www.diag.uniroma1.it//challenge9/download.shtml.

[12] Z. Abul-Basher. Multiple-query optimization of regular path queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1426–1430. IEEE Computer Society, 2017.

[13] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 2013.

[14] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 139–150. IEEE Computer Society, 2003.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[17] M. K. Esfahani, P. Kilpatrick, and H. Vandierendonck. Exploiting in-hub temporal locality in spmv-based graph processing. In X. Sun, S. Shende, L. V. Kalé, and Y. Chen, editors, *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 42:1–42:10. ACM, 2021.

[18] J. L. Gersting. *Mathematical structures for computer science (3. ed.)*. Computer Science Press, 1993.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30. USENIX Association, 2012.

[20] W. Han, S. Lee, K. Park, J. Lee, M. Kim, J. Kim, and H. Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 77–85. ACM, 2013.

[21] G. Jeh and J. Widom. Scaling personalized web search. In G. Hencsey, B. White, Y. R. Chen, L. Kovács, and S. Lawrence, editors, *Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003*, pages 271–279. ACM, 2003.

[22] U. Kang, D. H. Chau, and C. Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *ICDE*.

[23] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[24] M. Kaufmann, M. Then, A. Kemper, and T. Neumann. Parallel array-based single- and multi-source breadth first searches on large dense graphs. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 1–12. OpenProceedings.org, 2017.

[25] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[26] A. Koschmieder and U. Leser. Regular path queries on large graphs. In A. Ailamaki and S. Bowers, editors, *SSDBM*, volume 7338, pages 177–194. Springer, 2012.

[27] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46. USENIX Association, 2012.

[28] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 666–677. IEEE Computer Society, 2012.

[29] X. Liu, Z. Wang, N. Wang, X. Li, B. Zhang, J. Qiao, Z. Wei, and J. Nie. An adaptive sharing framework for efficient multi-source shortest path computation. In C. Xing, X. Fu, Y. Zhang, G. Zhang, and C. Borjigin, editors, *Web Information Systems and Applications - 18th International Conference, WISA 2021, Kaifeng, China, September 24-26, 2021, Proceedings*, volume 12999 of *Lecture Notes in Computer Science*, pages 635–646. Springer, 2021.

[30] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[31] S. Lu, S. Sun, J. Paul, Y. Li, and B. He. Cache-efficient fork-processing patterns on large graphs. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1208–1221. ACM, 2021.

[32] 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 A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[33] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable distributed evaluation of batches of iterative graph queries. In C. Baru, J. Huan, L. Khan, X. Hu, R. Ak, Y. Tian, R. S. Barga, C. Zaniolo, K. Lee, and Y. F. Ye, editors, *2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9-12, 2019*, pages 349–358. IEEE, 2019.

[34] U. Meyer and P. Sanders. Delta-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[35] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP'13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471. ACM, 2013.

[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[37] P. Pan and C. Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*, pages 217–224. IEEE Computer Society, 2017.

[38] A. Parravicini, F. Sgherzi, and M. D. Santambrogio. A reduced-precision streaming spmv architecture for personalized pagerank on FPGA. In *ASPDAC*, pages 378–383. ACM, 2021.

[39] P. Peng, Q. Ge, L. Zou, M. T. Özsu, Z. Xu, and D. Zhao. Optimizing multi-query evaluation in federated RDF systems. *IEEE Trans. Knowl. Data Eng.*, 33(4):1692–1707, 2021.

[40] S. Qiu, L. You, and Z. Wang. Optimizing sparse matrix multiplications for graph neural networks. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 101–117. Springer, 2022.

[41] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *Proc. VLDB Endow.*, 10(3):121–132, 2016.

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

[43] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[44] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.

[45] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013.*

[46] S. Skiena. *The Algorithm Design Manual, Second Edition*. Springer, 2008.

[47] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos. Accelerating graph analytics by utilising the memory locality of graph partitioning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 181–190. IEEE, 2017.

[48] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, 2014.

[49] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, 2013.

[50] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 943–972. Elsevier and MIT Press, 1990.

[51] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang. FORA: simple and effective approximate single-source personalized pagerank. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 505–514. ACM, 2017.

[52] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J. Wen. TopPPR: Top-k personalized pagerank queries with precision guarantees on large graphs. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 441–456. ACM, 2018.

[53] C. Xu, A. Mazloumi, X. Jiang, and R. Gupta. SimGQ: Simultaneously evaluating iterative graph queries. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*, pages 1–10. IEEE, 2020.

[54] C. Xu, A. Mazloumi, X. Jiang, and R. Gupta. SimGQ+: Simultaneously evaluating iterative point-to-all and point-to-point graph queries. *Journal of Parallel and Distributed Computing*, 164:12–27, 2022.

[55] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, 2014.

[56] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *Proc. VLDB Endow.*, 9(7):564–575, 2016.

[57] H. Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–10, 2010.

[58] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 441–452. USENIX Association, 2018.

[59] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe. GraphIt: a high-performance graph DSL. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, 2018.

[60] J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen. GraphM: an efficient storage system for high throughput of concurrent graph processing. In M. Taufer, P. Balaji, and A. J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 3:1–3:14. ACM, 2019.

[61] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In K. Keeton and T. Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 301–316. USENIX Association, 2016.