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.

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-Instance 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
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 frontier-ranking model that abstracts multiple-instance processing into numeric computations on ranks instead of imperative instructions in edge functions. By varying vertex ranks, one can compose existing and new multi-instance algorithms and (b) ease of use and simplicity of graph frameworks. MITra 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 \( G \).

- **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].
- **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 \( \Delta \)-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].

**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 \( \mathcal{F} \).

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

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

### 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., $\text{rank}(u)$, is populated to $k$ ranks such that $\text{rank}(u, s_i)$ 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 $\text{track}(u)$ that is a boolean array of length $k$ such that $\text{track}(u, s_i) = \text{True}$ 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, $\mathcal{F}$ consists of frontiers created for the sources according to their ranks. $\mathcal{A}_M$ starts the traversal with the top frontier of $\mathcal{F}$, 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., $\text{track}(u, s_i) = \text{True}$, and updates rank$(u, s_i)$; (2b) $\mathcal{A}_M$ then assigns $v$, for each source $s_j$, to some frontier $F'$ according to rank$(v, s_j)$; if it is changed in step (2a): if there exists such $F'$ in $\mathcal{F}$ whose domain covers rank$(v, s_j)$, it adds $v$ to $F'$ and sets track$(v, s_j)$ for $F'$; otherwise it creates $F'$ with $v$ for $\mathcal{F}$, with track$(v, s_j) = \text{True}$ and track$(v, s_j) = \text{False}$ for all $j \neq i$.

### Sharing

Intuitively, $\mathcal{A}_M$ groups traversals at vertex $v$ from multiple sources $s_j$ as long as their ranks rank$(u, s_j)$ 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., track$(v, s_i) = \text{True}$ (step 2a)). At this point the traversals from the sources are aligned and the computations for the sources can be therefore shared.

### Table 1: Graph computations in frontier-ranking algorithms

<table>
<thead>
<tr>
<th>Traversals</th>
<th>$f(e)$</th>
<th>rank$(u)$</th>
<th>$\delta$</th>
<th>$\mathcal{F}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>links</td>
<td># of rounds in which $v$ is visited</td>
<td>1</td>
<td>list</td>
</tr>
<tr>
<td>Reachability</td>
<td>bits</td>
<td>vid ID of vertex $v$ in data graph $G$</td>
<td>1</td>
<td>list</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>distance</td>
<td># of rounds in which $v$ is visited</td>
<td>1</td>
<td>list</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>distance</td>
<td>vid ID of vertex $v$ in data graph $G$</td>
<td>1</td>
<td>heap</td>
</tr>
<tr>
<td>$\Delta$-stepping</td>
<td>distance</td>
<td>tentative distance to the source; $\text{ans}(e) = \min(\text{ans}(a) + w(a, e), \text{ans}(b))$</td>
<td>$\Delta$</td>
<td>heap</td>
</tr>
<tr>
<td>PPR</td>
<td>pagerank</td>
<td># of rounds in which $v$ is visited</td>
<td>1</td>
<td>list</td>
</tr>
<tr>
<td>SpMV</td>
<td>spmv</td>
<td># of rounds in which $v$ is visited</td>
<td>1</td>
<td>list</td>
</tr>
</tbody>
</table>

*1 we detailed implementations of $f(e)$ in Section 4.1 and Section 5*
As shown in Fig. 1(b), after exploring $v_1$ in $F_1$ in round 1, $A_M$ assigns $v_0$ to (i) $F_0$ for both sources $s_1$ and $s_2$ since $\text{rank}(v_0, s_1)$ and $\text{rank}(v_0, s_2)$ are in $[n - \delta, (n + 1) - \delta]$, and (ii) $F_0$ for $s_4$ since $\text{rank}(v_0, s_4) \in [0, \delta)$. When $A_M$ fetches $F_0$ and explores $v_0$ of $F_0$ in round 2, the traversals and computations for $s_1$ and $s_2$ are then shared at $v_0$.

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 $A$ with specific in Table 1 for BFS (row 1), i.e., by setting $r$ as $\text{rank}$, $F$ to list and $\delta$ to 1. With the same frontier specification, multi-instance BFS, denoted by $\text{MITra-BFS}$, can be captured by $A_M$ by populating vertex properties (e.g., $\text{rank}$) of $A$ 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 $v_2$) in Fig. 2(b). The trace of $\text{MITra-BFS}$ for $v_0$, $v_1$ and $v_2$, in the language of $A_M$, is depicted in Fig. 2(c).

Initially, $F$ is empty and the ranks for all vertices and all sources are $\text{undefined}$ except $\text{rank}(v_0, s_0)$, $\text{rank}(v_1, s_1)$ and $\text{rank}(v_2, s_2)$ are set to $\text{#round}$, i.e., 0. $\text{MITra-BFS}$ creates frontier $F_0$ for $v_0$, $v_1$ and $v_2$ with frontier domain $[0, 1]$ covering their ranks. It sets $\text{track}(v_0, s_0)$, $\text{track}(v_1, s_1)$ and $\text{track}(v_2, s_2)$ to $\text{True}$ for $F_0$, and adds $F_0$ to $F$.

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

In total, $\text{MITra-BFS}$ 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 $\text{True}$ 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 $\text{MITra-BFS}$ of Example 1, exactly captures MS-BFS [48], the state-of-the-art multi-instance BFS algorithm. Similar to MS-BFS, one can cast the one-off MS-Dijkstra [57] as a frontier-ranking algorithm with vid as vertex ranks, $\delta = 1$, $F$ 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 $A_M$ as $\text{MITra-BFS}$. By simply changing its vertex ranks from $\text{#round}$ to vid and retaining everything else (row 2 of Table 1), we derive a new algorithm from MS-BFS, denoted by $\text{MITra-RCH}$, 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 $\text{#round}$ to vid, $\text{MITra-RCH}$ is able to reduce the cost of MS-BFS for sources $v_0(v_0)$, $s_1(v_1)$, $s_2(v_2)$ over $G$ in Fig. 2. The traversal trace of $\text{MITra-RCH}$ is shown in Fig. 2(d).

More specifically, $F$ is initially empty and the ranks for all vertices and all sources are $\text{undefined}$ except that $\text{rank}(v_0, s_0)$ is the vid of $v_0$, i.e., 0, $\text{rank}(v_1, s_1)$ is 1, and $\text{rank}(v_2, s_2)$ is 3 (assume w.l.o.g. that the vid of $v_1$ is 0). It then creates three frontiers $F_0 = \{v_0\}$, $F_1 = \{v_1\}$ and $F_2 = \{v_2\}$ such that the domain of $F_i (i \in \{0, 1, 3\})$ covers the vid of $v_i$. It then sets track$(v_0, s_0)$, track$(v_2, s_2)$ and track$(v_3, s_3)$ to $\text{True}$. It then adds the three frontiers to $F$ as a list $F_0 \rightarrow F_1 \rightarrow F_2$.

In round 1, as shown in row 1 in Fig. 2(d), $\text{MITra-RCH}$ pops out the “top” frontier from $F$, 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., \(r \text{ rank}(v_1, s_0)\), from \textit{undefined} to 1 since only \(\text{track}(s_0, s_0)\) is \textit{True}. It then assigns \(v_1\) for source \(s_0\) to new frontier according to the updated \(r \text{ rank}(v_1, s_0)\), which is \(F_1\) that already exists in \(F\); hence, it simply sets track\((v_1, s_0)\) to \textit{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 \(F\). So it creates \(F_2\) for \(v_2\) and appends \(F_2\) to \(F\), which is now \(F_1 \rightarrow F_3 \rightarrow F_2\).

Along the same lines, it pops and explores \(F_1\) from \(F\) in round 2 (row 2 in Fig. 2(d)). This continues until \(F\) 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 \(F\) column of Fig. 2(d). For instance, \((v_2, v_2)\) is accessed only once for all three sources by MITra-RCH in round 3 as the track of \(v_2\) for all three sources are \textit{True} for \(F_3\), while it is accessed three times by MS-BFS, one for source \(s_1\) in round 1, one for source \(s_2\) in round 2, and one for source \(s_0\) in round 3 (Fig. 2(e)).

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

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

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, \(\text{rank}_1()\) and \(\text{rank}_2()\). We say that \(\text{rank}_1\) is more \textit{sharing-permissive} if for any vertex \(u\) of \(G\), any sources \(s\) and \(s'\), the probability of \(\text{rank}_1(u, s) = \text{rank}_1(u, s')\) is no lower than that of \(\text{rank}_2(u, s) = \text{rank}_2(u, s')\). That is, \(\text{rank}_1\) is more permissive than \(\text{rank}_2\) if vertices are more likely to be ranked into the same frontier for the same source with \(\text{rank}_1\).

With the same edge function and frontier specification, i.e., \(F\) and \(\delta\), a frontier-ranking algorithm with vertex ranks that are more sharing-permissive will have a higher degree of sharing across instances. This has the following immediate implications.

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

\(2)\) \(\#\text{round} \leq \#\text{ans}\). Algorithms with \(\#\text{round}\) as ranks usually have better sharing than those use \(\text{ans}\), since vertices \(u\) visited by sources \(s\) and \(s'\) in the same round will be assigned with the same \(\#\text{round}\), i.e., \(\text{rank}(u, s) = \text{rank}(u, s')\), irrelevant of edge functions. In contrast, with \(\text{ans}\) 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.

\(3)\) \(F\) and \(\delta\). Frontier parameters \(F\) and \(\delta\) are also relevant. Indeed, if the domain of \(\text{ans}\) is small and \(\delta\) is large, \(\text{ans}\) can have more sharing than \(\#\text{round}\) since \(u\) may be ranked to the same frontier for \(s\) and \(s'\) by \(\text{ans}\) 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 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 \textit{model simulation}. Specifically:

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

Note that, if \(M\) optimally simulates \(M'\), it also practically simulates \(M'\), but not vice versa. Intuitively, if \(M\) optimally simulates \(M'\) but \(M'\) cannot simulate \(M\), then \(M\) is strictly more expressive than \(M'\). 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 \textit{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.
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 $\delta$ and frontier structure $F$, 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 $v$, 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 MPI 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, $F$ and $\delta$, MITra can express various multi-instance graph computations.

MITra programs. We show how to write MPI 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_Ore, 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] $\lor$ ans[v][i] if track[u][i] is True.

Shortest path. Similarly, multi-instance shortest path (SSSP) algorithms, e.g., Bellman-Ford, Dijkstra and $\Delta$-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, v) 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_Ore 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 $\mathcal{FM}$ (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 $\mathcal{F}$ from sources according to their ranks (line 5). It then iteratively explores frontiers in $\mathcal{F}$ via procedure MTraverse, by following step (2) of model $\mathcal{FM}$ (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 s₁, ..., sₖ.
Output: for each v ∈ V and i ∈ [1, k], the answer of v for sᵢ.
1 #define F frontier_structure, δ frontier_width;
2 #define type rank, type ans;
3 #define type X, type Y;
4 initialize vertex properties;
5 initialize F from s₁, ..., sₖ;
6 F ← F.pop(); // frontier for round i
7 while F ≠ nil do
8    [ F ← MTraverse(F, EdgeFunc);
9    return ans of all vertices;

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

Procedure UpdateF(v, pre_ans)
17    if [pre_ans[i] = nil then
18        update_rank(v, rank[i]); // built-in rank updates in MITra library
19        nFids ← mitra_Div(v, rank[v], δ);
20        foreach i ∈ [1, k] and ai ∈ [1, δ] do
21            if F[nFids[i]] = nil then
22                F[nFids[i]] ← v;
23            F.addFrontier(F[i], F[i].index ← nFids[i]); // F can now be referenced as F[nFids[i]]
24            F[nFids[i]].track[v][i] ← 1;
```

out a new frontier from 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[u][i]. This avoids unnecessarily putting v to future frontiers for sᵢ. It does this by comparing the values of ans[u] before and after the invocation of EdgeFunc, via MITra operator mitra_Rank (line 18) and updates rank[v][i] for sources sᵢ of which ans[u][i] are changed (line 20). It also decides frontiers to which v needs to be assigned for sᵢ, via mitra_Div (line 21).

2. It keeps track of traversal by maintaining track[v][i] for each source sᵢ when v is assigned to frontiers in F for future iterations (lines 22-27). When it assigns v to a new frontier Fₙ for sᵢ, it first removes v from the existing frontier for sᵢ by setting its track[v][i] to 0 (line 23). Then it adds v to Fₙ by setting track[v][i] to 1 for Fₙ (lines 27), if such Fₙ is not yet F, it creates 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 A-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| × k such that F[v][i] is True if v ∈ 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 sᵢ 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 MITa algorithms. (1) One may optionally specify a vertex function fᵢ(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, e.g., Intel intrinsics [2] that support vectorized mask operations, by treating track[u] as mask, e.g., mitra_AddX(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_AddX(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.
Example 4: Consider a weighted graph $G$ in Fig. 3(a), which has (1) a connected component $C_1$ with vertices $s_0, s_1, \ldots, s_{k-1}, v_0, v_1, \ldots, v_l$, (2) connected component $C_2$ with vertices $v_{l+1}, v_{l+2}, \ldots, v_n$, 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 $\text{track}[v_i]_F$ 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., $\text{track}[v_i]_F[1] = \text{True}$. Upon visiting edge $(v_i, v_j)$, the edge function for SSSP (i.e., Fig. 3(b)) is invoked for updating $\text{ans}[v_j]$ using $\text{ans}[v_i]$ via MITra operations masked by $\text{track}[v_i]$. One can verify that (1) without TrackFree, $\text{ans}[v_j]$ is updated to Fig. 3(c), i.e., only $\text{ans}[v_i][1]$ is progressed from 7 to 4 (blue, underlined); in contrast, (2) with TrackFree, all $k$ values of $\text{ans}[v_j]$ are updated as shown in Fig. 3(d). The immediate effect of TrackFree is that changes of all $k$ values of $\text{ans}[v_j]$ 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 $\text{ans}[v][i]$ 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 $x^{t+1} = Ax^t = \sum_{i=1}^{n} A_i \otimes x^t$, 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; $\otimes$ and $\times$ are algorithm-specific semiring operators. To reduce notation, we simply adopt the standard $+$ and $\times$ for $\otimes$ 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 $\text{ans}_\text{cur}$ and $\text{ans}_\text{next}$, where $\text{ans}_\text{cur}$ is the value of $x$ from the last round and $\text{ans}_\text{next}$ is its value updated by EdgeFunc in the current round. Initially, $\text{ans}_\text{cur}$ is set to according to $x_j (j \in [1, k])$ and $\text{ans}_\text{next}$ is 0 for all vertices and all instances (line 2).

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

$$p^{t+1} = \alpha P p^t + (1 - \alpha) \cdot c,$$

where $\alpha$ is the damping factor, $X = AD^{-1}$, where $D$ is the out-degree matrix of $G$ (a diagonal matrix with the number of out-degree of each vertex), $p^0$ is the PPR values at iteration $t$, and $c$ is an identity vector ($c_i[v] = 1$ when $v = s$ 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
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 implementations 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-Bellm, MiTra-Dijkstra, 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 MPI 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 $\Delta$-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.

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-Bellm, MiTra-Dijkstra, 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-SpMV (Reachability), Ligra-Bellm, kDijkstra, k-$\Delta$-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
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-Dijkstra vs. one-off MS-Dijkstra is similar. On average MS-Dijkstra is only 3.35% faster than MITra-Dijkstra.

(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. 4.

(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 4.28x 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-Dijkstra, 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.
SIMP. 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 $2^{25}$ 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 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)
We categorize related work as follows. (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 $k = \frac{M - |V|}{p|s|v|siz}\circ(T)$. One can divide the sources into $\lfloor \frac{|S|}{k} \rfloor$ 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 multicore 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 $\#\text{round}$ as rank, $F$ 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-Dijkstra 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. Zhao and Buneman are supported by the Edinburgh-Huawei Joint Lab.
REFERENCES


