Automating Vectorized Distributed Graph Computation

WENYUE ZHAO, University of Edinburgh, UK
YANG CAO, University of Edinburgh, UK
PETER BUNEMAN, University of Edinburgh, Uk
JIA LI, Edinburgh Research Center, Central Software Institute, Huawei, UK
NIKOS NTARMOS, Edinburgh Research Center, Central Software Institute, Huawei, UK

Multi-instance graph algorithms interleave the evaluation of multiple instances of the same algorithm with different inputs over the same graph. They have been shown to be significantly faster than traditional serial and batch evaluation, by sharing computation across instances. However, writing correct multi-instance algorithms is challenging; and in this work, we describe AutoMI, a framework for automatically converting vertex-centric graph algorithms into their vectorized multi-instance versions. We also develop an algebraic characterization of algorithms that can benefit best from multi-instance computation with simpler and faster streamlined vectorization. This allows users to decide when to use such optimization and instruct AutoMI to make the best use of SIMD vectorization. Using 6 real-life graphs, we show that AutoMI-converted multi-instance algorithms are 9.6 to 29.5 times faster than serial evaluation, 7.1 to 26.4 times faster than batch evaluation, and are even 2.6 to 4.6 times faster than existing highly optimized handcrafted multi-instance algorithms without vectorization.

CCS Concepts: • **Information systems** → **Graph-based database models**; • **Software and its engineering** → **Source code generation**.

Additional Key Words and Phrases: Graph computation, SIMD vectorization, Auto vectorization, Algebraic characterization

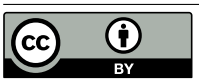
ACM Reference Format:
Wenyue Zhao, Yang Cao, Peter Buneman, Jia Li, and Nikos Ntarmos. 2024. Automating Vectorized Distributed Graph Computation. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 254 (December 2024), 27 pages. https://doi.org/10.1145/3698833

1 INTRODUCTION

Multi-instance graph algorithms have found widespread use in social analysis [30, 35, 54], web search [33, 44, 61] and bioinformatics [14, 37, 48], where one needs to run the same query multiple times with different source vertices as inputs. Multi-instance algorithms interleave their evaluation in order to exploit computation sharing across source vertices. In addition, they can capitalize on vectorization as the same algorithm is invoked for all source vertices.

These make multi-instance algorithms significantly faster than traditional serial evaluation that process the queries one by one. For example, multi-instance breadth-first search (BFS) has been shown an order of magnitude faster than serial evaluation for 256 sources over real-life graphs with million vertices [54]; similarly, a multi-instance implementation of the Bellman-Ford

Authors' addresses: Wenyue Zhao, University of Edinburgh, UK, wenyue.zhao@ed.ac.uk; Yang Cao, University of Edinburgh, UK, yang.cao@ed.ac.uk; Peter Buneman, University of Edinburgh, Uk, peter.buneman@ed.ac.uk; Jia Li, Edinburgh Research Center, Central Software Institute, Huawei, UK, jiali11@huawei.com; Nikos Ntarmos, Edinburgh Research Center, Central Software Institute, Huawei, UK, nikos.ntarmos@huawei.com.



ACM 2836-6573/2024/12-ART254
https://doi.org/10.1145/3698833

Table 1. Running time of multi-instance graph algorithms: (1) PowerLyra [20] processes sources one by one using single-instance algorithms; Quegel [59, 64] uses batching to process multiple sources concurrently; MultiLyra [40] uses hand-crafted implementation of multi-instance algorithms from its authors; AutoMI directly runs its auto-converted GAS programs using PowerLyra. (2) `GC`: greedy graph coloring [24]; `CF(SGD)`: collaborative filtering with SGD [31]; `SpMV`: Sparse Matrix-Vector multiplication [21]. (3) ✗: algorithm is not available. (4) We used 256 sources over graph `LiveJournal` [5] with 4.8M vertices and 69M edges, and `MovieLens` [6], a bipartite graph with 0.16M vertices and 20M edges (for `CF` only).

methods	`Boolean BFS`	`Int. BFS`	`SSSP`	`SpMV`	`PageRank`	`GC`	`CF(SGD)`
PowerLyra	297.28s	525.94s	404.61s	742.15s	873.17s	609.62s	749.78s
Quegel	247.42s	355.69s	315.43s	✗	✗	✗	✗
MultiLyra	35.42s	58.57s	55.96s	✗	✗	✗	✗
AutoMI	**10.69s**	**32.02s**	**13.71s**	**73.55s**	**105.93s**	**81.35s**	**121.52s**

algorithm for single-source shortest path (SSSP) has been shown to give a 5-fold improvement over a single-instance implementation for 64 source vertices [40].

While a well developed multi-instance algorithm can bring significant performance boost, a caveat is that they are notoriously hard and delicate to design and implement. For instance, to get the above speedup, multi-instance BFS takes over 550 lines of highly engineered C++ code to just implement its main algorithmic logic [7].

This is because the performance boost relies on carefully aligned execution of different instances to share graph traversal and computation costs, which requires significant engineering effort to implement properly. In addition, hand-rolled SIMD vectorization adds a further layer of challenges in implementing efficient multi-instance graph algorithms. While there has been recent frameworks to assist developers to write multi-instance algorithms [34, 40], they still require carefully crafted implementation from users to address these challenges and to do so with specialized programming models.

AutoMI. In response to the challenge of designing and implementing effective multi-instance algorithms, we present AutoMI, a framework that automatically converts traditional single-instance algorithms into their vectorized multi-instance version. AutoMI relieves developers from the burden of writing delicate multi-instance algorithms and achieves superior performance through vectorization.

AutoMI targets vertex-centric algorithms [36, 39, 41] written in the GAS (Gather-Apply-Scatter) programming model as promoted by major distributed graph computing frameworks [20, 24] for their simplicity. Given any GAS program that implements a traditional single-source graph algorithm *e.g.,* BFS or SSSP, AutoMI generates a GAS program correctly implementing the corresponding multi-instance version that can run on any generic GAS system.

Moreover, multi-instance algorithms generated by AutoMI are highly efficient, up to an order of magnitude faster than serial evaluation of single-instance algorithms over the same GAS engine, and are even faster than handcrafted multi-instance algorithms implemented with a dedicated programming framework. As shown in Table 1, it takes PowerLyra [20], the successor of PowerGraph [24], 297.28s to process Boolean BFS with 256 sources using its built-in single-instance BFS. This is reduced to 10.69s by just running the multi-instance BFS converted by AutoMI on PowerLyra, improving the latter by 27.8 times. It also has similar performance advantages over Quegel [59, 64], a query batching framework that evaluates multiple sources concurrently. Better still, it is even 3.3 times faster than the handcrafted multi-instance BFS in MultiLyra [40] by virtue of its embedded vectorization in the generated algorithms; similarly for other algorithms (see more in Section 7).

Automating vectorization. The natural idea of converting a single-instance algorithm into its multi-instance version is to populate vertex properties for all sources and vectorize associated operations. However, we show that such direct vectorization of GAS programs can cause incorrect answers as it critically relies on the assumption that each time the GAS program fires at a vertex v, all sources are visiting v. To remedy this, AutoMI automatically keeps track of the traversal progress of all sources and assures that the converted algorithm only progresses those sources that would visit v if they each run independently. To do this, AutoMI first generates an abstract syntax tree (AST) of the single-instance GAS algorithm and annotates it with track, a carefully deduced annotation that helps trace the progress of sources. It then generates GAS program from the annotated AST, converting annotations into SIMD (single-instr.-multi-data) masks that guard against incorrect vectorization of vertex operations. We prove that, with track, AutoMI always produces correct multi-instance GAS programs that benefit from vectorization.

Track-free vectorization via idempotence. While track guarantees that multi-instance GAS algorithms generated by AutoMI are always correct, there exist algorithms for which track is not necessary. For such algorithms, users can instruct AutoMI to generate streamlined track-free vectorized GAS programs that are significantly simpler and faster. For instance, `Boolean BFS` in Table 1 is track-free while `Int. BFS` is not, which is consistent with the speedup of AutoMI over PowerLyra: 27.8 times for `Boolean BFS` and 16.4 times for `Int. BFS`.

A critical problem is to decide when it is safe for AutoMI to use track-free vectorization. To this end, we extend linear algebra to allow users to abstract algorithmic logic of the Apply and Scatter functions of their GAS programs, independently, in the extended algebra. We then develop a formal characterization of track-free vectorization of single-instance GAS algorithms by reducing to two *algebraic idempotence* properties of their algebra abstraction, allowing users to decide when to enable track-free vectorization in AutoMI.

Putting this together, AutoMI automatically generates a variety of multi-instance GAS algorithms and benefits from track-free vectorization when users decide it is safe for their algorithms, yielding 9.6 to 29.5 times of speedup over real-life graphs (see Sections 6-7).

Contributions. In summary, we make the following contributions:

- We develop AutoMI that, given a single-instance GAS algorithm $\mathbf{A}$, automatically generates a multi-instance GAS algorithm $\mathbf{A}_m$ for $\mathbf{A}$ that can run over existing GAS systems (Section 3).
- We propose `TrackFree`, an optimization for a class of algorithms of which $\mathbf{A}_m$ can be made significantly simpler and faster (Section 4).
- We give an algebraic characterization of $\mathbf{A}$ to decide when to use `TrackFree` in AutoMI (Section 5).
- We show how AutoMI generates multi-instance programs for a variety of algorithms (Section 6).
- We implement AutoMI and evaluate its effectiveness (Section 7).

Related Work. There is extensive related research.

Multi-instance algorithms. These have been developed for SQL [50, 51], XQuery [16], SPARQL [33, 44], subgraph [48] and regular path query (RPQ) [14]. They identify common sub-queries and re-use them to answer multiple queries. For graph traversal, multi-instance algorithms are developed to interleave the evaluation of the same query with multiple sources, yielding much higher efficiency [30, 34, 54, 61]. Among them, MITra [34] is a generic framework for composing single-thread multi-instance graph traversal algorithms. In contrast, AutoMI automatically converts single-instance GAS algorithms into optimized multi-instance GAS algorithms.

Distributed graph frameworks. There are a number of vertex-centric distributed graph computation frameworks [2, 20, 24, 29, 36, 39], and we refer readers to recent surveys [15, 27, 38, 41, 58, 63] for a detailed coverage. They target single-instance graph algorithms and do not help with multi-instance computation. AutoMI is orthogonal to and complements these, directly benefiting GAS systems.

Also related is the study of distributed [40, 57, 67] and parallel [37, 43, 62, 65, 68] concurrent iterative job processing. They optimize groups of jobs via scheduling and partitioning to improve efficiency [19, 43, 60, 62, 67] and cache hit rate [37]. They are orthogonal to multi-instance algorithms that share computation across jobs via interleaved evaluation. They also require users to re-implement algorithms via their interface as opposed to automated conversion that AutoMI provides.

Quegel [59, 64], a distributed system for batched graph query processing, allows users to specify queries as single-instance vertex-centric algorithms using the programming model of Pregel [39]. It employs batch scheduling to execute multiple query instances concurrently. Compared to AutoMI, Quegel's batching technique misses out on vectorization. Moreover, Quegel does not integrate computation and communication across query instances.

Closer is MultiLyra [40] that extends PowerLyra to enable users to program batched graph query evaluation of multiple instances. Compared to AutoMI, MultiLyra requires manual implementation of multi-instance programs and also misses out on vectorization.

Graph algorithm compilation. Many domain-specific languages have been proposed for program synthesis for graph algorithms [28, 46, 47, 66]. They obtain high-level algorithm specifications from users and generate efficient algorithm implementation. However, they do not produce multi-instance algorithms. Closer to our work is SAMA [55], which transforms graph algorithms into multi-snapshot algorithms optimized for concurrent execution over multiple snapshots of temporal graphs. In contrast, AutoMI automatically converts single-instance GAS algorithms into multi-instance programs for computation sharing over the same non-temporal graph.

Semiring frameworks. There has been extensive research on semiring frameworks for modeling provenance [17, 25, 26] and expressing queries [13, 23, 42]. The algebraic characterization proposed in Section 5 builds upon the machinery from prior art to, however, capture `TrackFree` property of GAS programs rather than queries, to help users decide when it is safe to enable `TrackFree`.

2 PRELIMINARIES

Graphs. A graph is denoted by $G(V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. We consider directed graphs possibly with edge weights, *i.e.,* there exists a function w that assigns each edge $e \in E$ a property $w(e)$ from a domain, *e.g.,* integers.

Queries and algorithms. We consider traversal queries over graph $G(V, E)$. More specifically, a query $Q(s)$ takes as input a source vertex $s \in V$ and returns, for each vertex $v \in V$, an answer value $\mathsf{ans}(v)$ according to the query semantics of Q.

For example, a Boolean breadth-first search (BFS) query $Q_{\texttt{bBFS}}(s)$ asks, for each vertex v of G, whether v is reachable by a breadth-first search starting from s; an Integer BFS query $Q_{\texttt{iBFS}}(s)$ instead computes the level of each v of G in the breadth-first search iteration starting from s; a distance query $Q_{\texttt{SSSP}}(s)$ asks the distance from s to each vertex v of G. More examples are discussed in Section 6.

We refer to algorithms that evaluate such traversal queries as *single-instance* algorithms (denoted by $\mathbf{A}$) since they take as input a single source vertex and compute answers of all vertices for the query.

Multi-instance algorithms. A *multi-instance* algorithm $\mathbf{A}_m$ for traversal query Q answers Q for multiple source vertices. Specifically, given graph G and source vertices $s_1, \ldots, s_k$ of G, $\mathbf{A}_m$ for Q computes the answers to $Q(s_1), \ldots, Q(s_k)$ in G. We opt not to use "multi-source" to distinguish from algorithms that return a single aggregated answer for multiple sources, *e.g.,* multi-source SSSP [35].

ALGORITHM 1: Single-instance Integer BFS

```
1  #define vertex_data { <int> ans, <bool> visited } ;          // vertex properties
2  #define msg { <int> ans } ;                                  // define message struct
3  Function Gather(vertex u, edge (u, v), vertex v)            // invoked at v (center vertex)
4      return msg                                               // sent from u by Scatter in previous round
5  Function Gather_Acc(vertex v, msg_acc, msg)                 // center vertex: v
6      msg_acc.ans ← Min (msg_acc.ans, msg.ans) ; // accumulate and aggregate msg in msg_acc; initially, msg_acc.ans is 0 if
         v is source and +∞ otherwise
7      return msg_acc;
8  Function Apply(vertex v, msg_acc)                           // center vertex: v
9      if NOT v.visited then
10         v.ans ← msg_acc.ans;
11         v.visited ← True;
12         changed ← True ;                                     // mark changed as True
13 Function Scatter(vertex v)                                  // center vertex: v
14     if changed then                                          // check if changed is True
15         msg.ans ← v.ans + 1 ;
16         Signal (msg) ;                                       // activating and sending msg to outward neighbors
```

A naive multi-instance algorithm for any query Q is to invoke a single-instance algorithm for each source, computing $Q(s_1), \ldots, Q(s_k)$ one by one; we refer to this as a serial evaluation. However, this does not exploit the huge space of computation sharing across sources. Indeed, there is substantial work in developing "real" multi-instance algorithms that interleave the evaluation of Q for multiple sources, yielding much higher efficiency [30, 34, 54, 61].

Vertex-centric algorithms. These are algorithms in which the same code is associated with each node in a graph. Evaluation is simple and proceeds in parallel. At each step, each of a set of designated nodes reads values from its immediate neighbors, it processes that data, to obtain values that are sent to its neighbors for the next round. They are widely adopted by distributed graph computing systems, *e.g.,* Pregel [39], GraphLab [36]. We refer readers to [15, 27, 38, 41, 58, 63] for comprehensive surveys on vertex-centric algorithms.

Populating algorithms. In contrast to designing effective multi-instance graph algorithms as prior work does, we take a different approach, to automatically convert single-instance algorithms **A** into their multi-instance version $\mathbf{A}_m$, which we refer to as populating **A**.

3 VECTORIZING GRAPH ALGORITHMS

We show that populating algorithms is doable, automatically, for vertex-centric graph algorithms.

3.1 Multi-instance GAS Algorithms

We focus on vertex-centric algorithms implemented with the GAS (Gather-Apply-Scatter) programming model, as promoted by *e.g.,* PowerGraph [24] and PowerLyra [20] for its simplicity. We remark that our methods in principle can be adapted for other models [39].

GAS via an example. To see what GAS programs look like and how they run, we use Integer BFS (recall Section 2) as an example. Algorithm 1 depicts a textbook Integer BFS in GAS.

Algorithm 1 abstracts the vertex computation logic of BFS and is invoked by a GAS system in synchronized rounds for each *active* vertex independently. It consists of (a) a preamble (lines 1-2) that defines vertex properties, *i.e.,* ans and visited for BFS, and a mandatory message structure

ALGORITHM 2: Multi-instance Integer BFS

```
1  #define vertex_data { array <int> ans, bitvec <bool> visited } ;
2  #define msg { array <int> ans } ;
3  Function Gather(vertex u, edge (u, v), vertex v)
4      return msg ;
5  Function Gather_Acc(vertex v, msg_acc, msg)
6      msg_acc.track ← simd_Or (msg_acc.track, msg.track) ;
7      msg_acc.ans ← simd_Min (msg_acc.ans, msg.ans) ;
8      return msg_acc;
9  Function Apply(vertex v, msg_acc)
10     mask ← simd_mask_Neg (msg_acc.track, v.visited) ;
11     v.ans ← simd_mask_Set (mask, v.ans, msg_acc.ans) ;
12     v.visited ← simd_mask_Set (mask, v.visited, 1) ;          // 1: constant True vector
13     changed ← simd_mask_Set (mask, changed, 1) ;
14 Function Scatter(vertex v)
15     mask ← simd_mask_Set (msg_acc.track, mask, changed) ;
16     msg.ans ← simd_mask_Add (mask, v.ans, 1);
17     if NOT simd_All_Zero (mask) then                          // if changed is True for some source
18         msg.track ← simd_Set (msg.track, mask) ;
19         Signal (msg) ;
```

msg that captures the communication between adjacent vertices, and (b) functions that specify the vertex computation logic (lines 3-16). In each round, it is invoked on each of the active vertices of G in parallel, where a vertex v is active if there exists some vertex v' that signals v (line 16) in the previous round; initially, only the source s is active.

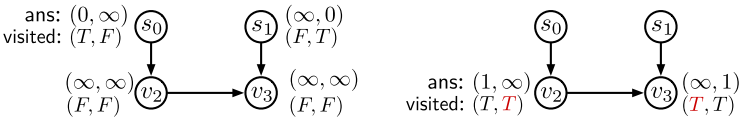
More specifically, a GAS vertex program consists of 4 functions, executed in 3 stages with synchronization barrier in between:

(1) *Gather* phase. Function `Gather` is invoked at each active vertex v to pull `msg` from inward neighbor vertices (line 4). The messages from all neighbors are aggregated at v as `msg_acc` via `Gather_Acc` (lines 6-7). For BFS, `msg` from vertex u is the BFS depth of u plus 1, assigned by `Scatter` function (line 15) in the previous round. Function `Gather_Acc` is a binary operation function on `msg` and `msg_acc` that is commutative and associative. With these properties, aggregating on this function can itself be parallelized, which is important when the number of neighboring vertices is large.

(2) *Apply* phase. `Apply` is then invoked at v using vertex properties and `msg_acc` of v only (line 8). For BFS, it checks if v is visited: if not (*i.e.,* v.visited is *False*), it updates the BFS depth of v with `msg_acc` (line 10), marks v as visited (line 11) and its answer as changed (line 12). Here `changed` is a reserved local variable commonly used in GAS programs to indicate if vertex properties at v is changed in the round; it is reset to *False* at the start of `Apply` phase in each round.

(3) *Scatter* phase. Finally, v decides whether to activate outward neighbors for the next round via `Scatter` (line 13). For BFS, v activates neighbors only when its answer was just updated by `Apply`, *i.e.,* `changed` is *True* (line 12). If so, v assigns its answer, *i.e.,* BFS depth, plus 1 to `msg` (line 15) and signals neighbors to be active for the next round (line 16), which will pull `msg` from v in *Gather* phase.

A GAS system would invoke Algorithm 1 as the vertex function of BFS at all active vertices, round by round, until no vertex is signaled (active); it then returns the ans property of all vertices. By



(a) before round 1 (initial state) **(b)** after round 2

Fig. 1. Data graph for Example 1 (sources: s_0 and s_1)

signaling active vertices for each round, systems need only invoke programs for vertices that require computation instead of blindly running at all vertices, making them practical and scalable [39, 41, 59].

Why multi-instance GAS? There are three reasons that we may want multi-instance GAS algorithms when handling multiple sources.

(1) A multi-instance GAS algorithm often requires fewer rounds to complete than answering sources one by one with a single-instance GAS algorithm. For BFS, we can signal all sources initially and then the BFS search would start from each source logically in parallel.

(2) Moreover, they enable us to group the vertex computations for different sources. This increases data locality and opens an avenue for vectorization, benefiting from *e.g.,* SIMD on modern CPUs.

For BFS, assume that we have 8 sources visiting v. Then v.ans + 1 of line 15 in Algorithm 1 can be vectorized using Intel SIMD intrinsics [3] as `_mm256_add_epi32(v_ans, _mm256_set1_epi32(1))`, where `v_ans` is a vector of 8 integers such that `v_ans`$[i]$ is the BFS depth from the i-th source. This allows us to share computations across BFS instances of different sources via *e.g.,* SIMD.

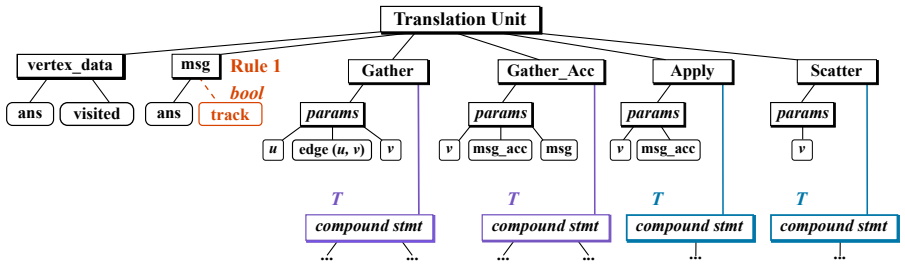
Why does SIMD vectorization not work? One may think that writing multi-instance algorithms in GAS is just to populate vertex properties into vectors and vectorize operations on vertex properties via *e.g.,* SIMD (which we refer to as streamlined vectorization). It is however much more than that.

In fact, blindly applying streamlined vectorization effectively assumes that all sources are visiting the active vertex v when a GAS function fires at v, which does not hold in many cases. As a result, a multi-instance GAS algorithm produced by streamlined vectorization could be incorrect as a vertex that is "activated" for only one source would be treated as if it were activated for all.

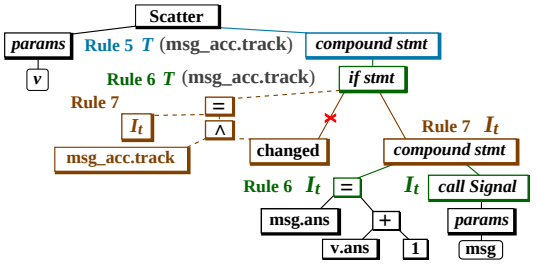
A counterexample: vectorized BFS. Consider BFS in Algorithm 1. By populating vertex property and vectorizing operations via branchless programming [56], we obtain a multi-instance BFS with streamlined vectorization as *Algorithm 2 without the part shadowed in gray.* Each vertex property is now a vector and the control flow of `Apply` in Algorithm 1 is vectorized as lines 10-13 of Algorithm 2 via *masked SIMD* [3, 45, 56]; similarly for lines 14-16 of Algorithm 1.

Here a masked SIMD is of the form `simd_mask_OP` (`mask`, `data`), where `mask` is a bitvec of same length as `data` such that masked operation `OP` only applies to `data`$[i]$ when `mask`$[i]$ is *True*. For instance, `simd_mask_add` (`mask`, `ans`, 1) can be implemented using Intel SIMD intrinsics [3] as `_mm256_mask_add_epi32(mask, ans, _mm256_set1_epi32(1))`, such that 1 is added to `ans`$[i]$ for source s_i only if `mask`$[i]$ is *True*. Even with masked SIMD, Algorithm 2 (without shadowed part) is however still incorrect for BFS. Below we give a counterexample as a proof.

Example 1: Consider Fig. 1(a), where we start BFS from two sources s_0 and s_1 via streamlined vectorization, using Algorithm 2 without the shadowed part. Initially, `ans`$:(0,+\infty)$ of s_0 shows that the BFS depth for s_0 (resp. s_1) is 0 (resp. $+\infty$); similarly for visited. At the end of round 1, v_2 and v_3 are signaled by their neighbor s_0 and s_1 (line 19), respectively, which then update their `ans` and visited as depicted in Fig. 1(b) via `Apply` in round 2. This however gives us incorrect visited for both v_2 and v_3 as highlighted in red in Fig. 1(b), which prevents v_3.`ans` from being correctly updated to (2, 1) in round 3.



(a) The root portion of the annotated AST for Algorithm 1.



(b) `Scatter` portion of the annotated AST.

Fig. 2. Annotated AST of Algorithm 1 (dashed lines denote added edges during annotation)

The incorrect update in round 2 is caused by v_2 and v_3 being treated as visited by both sources s_0 and s_1, while in fact only s_0 visits v_2 and s_1 visits v_3. Such a mistake in traversal logic occurs because the streamlined vectorization (line 10 of Algorithm 2 without shadowed part) sets the SIMD mask for both sources to *True*, forcing visited flipped to *True* for both sources by line 12. □

Example 1 shows that simply populating vertex properties and vectorizing operations can lead to incorrect results. Worse still, vertex properties of v do not tell `Apply` which sources are visiting v when it fires at v; similarly for other GAS functions. In fact, the correct multi-instance BFS derived from Algorithm 1 should be the complete Algorithm 2 including the part shadowed in gray, which is a far cry from a streamlined vectorization of Algorithm 1.

We remark that Bellman-Ford (Algorithm 4, Section 6) on unweight graphs could also compute BFS depths, and warrants correct results with streamlined vectorization. This indicates that the correctness of streamlined vectorization is algorithm dependent, which adds to users' confusion. Moreover, there are queries for which streamlined vectorization may not work at all. For instance, we are not aware of any multi-instance Collaborative Filtering algorithm with streamlined vectorization.

Motivated by the delicacy of multi-instance GAS algorithms and tricky decision on streamlined vectorization, we develop AutoMI, a framework to automate the conversion from single-instance to vectorized multi-instance GAS algorithms (Section 3.2), and give a characterization (Section 5) to aid the decision of whether streamlined vectorization can be used safely.

3.2 Automating Vectorization

To deal with the subtlety of writing correct and efficient multi-instance algorithms, we develop AutoMI, a framework for converting a given single-instance GAS program (*e.g.,* Algorithm 1) into a provably correct multi-instance program (*e.g.,* Algorithm 2) that directly runs on any GAS system.

The key idea is to (a) deduce the exact sources that are visiting v when a GAS function is invoked in the presence of multiple sources, and (b) inject the information on-the-fly when vectorizing the

Table 2. Annotation rules: *each line on* RHS *of the rules denotes a node of the annotated AST with* T *as the label if* mark(T) *is applied.*

Rule 1 Message Structure `#define msg {...}`	⇒	`#define msg {...<bool> track}`
Rules 2-5 `Gather`, `Gather_Acc`, `Apply`, `Scatter` Functions Function Name (*args* ...) *stmt*	⇒	Function Name (*args* ...) mark(T) *stmt*
Rule 6 Compound Statement mark(ℓ) {$stmt_1 ... stmt_n$} // ℓ could be 'T' or new variable (rule 7)	⇒	mark(ℓ) $stmt_1$... mark(ℓ) $stmt_n$
Rule 7 IF Selection Statement mark(ℓ) **if** E **then** $stmt_t$ **else** $stmt_f$	⇒	$I_t \leftarrow \ell \wedge E$ /* I_t: a distinct variable*/ **if** I_t **then** mark(I_t) $stmt_t$ $I_f \leftarrow \ell \wedge \neg I_t$ **else** mark(I_f) $stmt_f$

input GAS program. Moreover, both (a) and (b) have to be done without knowing the algorithm logic or degrading the effectiveness and efficiency of the vectorized multi-instance computation.

To do this, AutoMI automatically deduces a "mask" vector called track to keep track of the progress of all the sources and serve as the mask of vectorized (SIMD) operations on vertex properties (recall masked SIMD from Section 3.1). It injects track into the vectorization process via `Scatter` and automatically maintains it without assuming any knowledge of the algorithm logic.

Roadmap. AutoMI does the auto-conversion in two steps.

- *Annotation step*: it first parses the input GAS program as an abstract syntax tree (AST) and then annotates the AST with a set of annotation rules in Table 2.
- *Code-generation step*: it then traverses the annotated AST and generates GAS code snippets by populating data structures and vectorizing operations with masks interpreted from annotations of the AST, via code generation rules in Table 3. This gives us an executable GAS program that correctly implements the multi-instance version of the input algorithm.

Step (1) AST annotation. AutoMI first obtains the AST representation of input GAS program via a compiler front-end. Figure 2a shows the AST of Algorithm 1. The root of AST represents the entire program, which has 6 children among which 2 encode the structure definitions, *i.e.,* vertex properties and `msg`, and the other 4 encode the GAS functions. Each function node is also the root of the AST of encoded function itself, which consists of arguments ('params' in Fig. 2a) and compound statements that constitute the function body.

AutoMI annotates the AST by traversing it from the root and checking if any of the *annotation rules* of Table 2 applies when it visits an AST node. The annotation rules either (a) attach a track label to an AST node or (b) adjust the structure of the AST.

There are 7 annotation rules (Table 2). Rule 1 instructs AutoMI to add a new child node, a Boolean member called track, to the `msg` node in AST. Rules 2-5 simply mark the body of the 4 GAS functions with a designated label 'T' (via mark(T)). Rule 6 propagates 'T' into the statements in the GAS functions. Rule 7 deals with **If** ... **Else** ... control flow, by adding two distinct variable names I_t and I_f to $stmt_t$ and $stmt_f$, respectively, as annotation labels. In the code-generation step next, I_t and I_f,

together with *'T'*, will be interpreted as the SIMD mask for vectorized operations. Note that in rule 6-7, annotation label ℓ could be *'T'* or new variable names introduced by annotation rule 7. Intuitively, label'T' is recursively assigned to all nodes except control flow statements which is dealt by rule 7.

Example 2: We show how AutoMI annotates the AST of Algorithm 1. As shown in Fig. 2a, AutoMI first extends the `msg` node with a child (track) via rule 1. It then annotates the body of all GAS functions with *'T'*. It traverses the subtree of each GAS function body and propagates annotations further down the AST. Figure 2b shows how AutoMI annotates the `Scatter` function. AutoMI first propagates *'T'* to `Scatter` function body via rule 5. It then visits function body root, *i.e.,* the "compound stmt" node, and identifies that rule 6 applies and further visits the "if stmt" node, to which rule 7 applies. According to rule 7, it adjusts the structure of AST as shown in Fig. 2b and attaches the deduced labels to the children of "if stmt" (colored in brown). Note that for `Scatter` of Algorithm 1, there is no $stmt_f$ branch as also shown in Fig. 2b. □

Step (2) Vectorized code generation. AutoMI produces a multi-instance GAS program of the input algorithm by conducting a depth-first traversal on the annotated AST, with code fragments generated and pieced together from bottom-up. At each node, AutoMI assembles and completes code fragments from child nodes, to form the code fragment of the current node. It returns the code fragment of the root of AST as the complete multi-instance GAS program.

Upon visiting an AST node, AutoMI generates GAS code via `Codegen`, a function that takes as input a (sub-)tree of the annotated AST rooted at the current node, and returns the corresponding GAS program. In the process, `Codegen` interprets the annotation labels, *i.e., 'T'* or distinct variable names (*e.g.,* I_t) introduced by the annotation of control flow, as SIMD masks for the vectorized operations.

`Codegen` is inductively defined by the code-generation rules in Table 3. It does the following.

Structure populating (rule 1 of Table 3). Assume we have k sources $s_1, \ldots, s_k$. `Codegen` first populates each vertex property p into a k-size array $\mathbf{p}$ such that $\mathbf{p}[i] (i \in [1, k])$ corresponds to property p for source s_i; similarly for Message structure `msg`.

In addition, `Codegen` adds a new variable called track, a `bool bitvec` of length k, to the definition of `msg`. It will be injected to the generated code by `Codegen` via masked SIMD, to automatically keep track of traversal progress of all sources.

Track-*guarded vectorization*. With track, `Codegen` generates a complete GAS program from the annotated AST, while warranting its correctness as a multi-instance version of the input algorithm.

(a) Rules 2-3: function generation. When AutoMI visits one of the 4 GAS function nodes, `Codegen` simply combines the code fragments of its left (*i.e.,* 'params') and right (*i.e.,* function body 'compound stmt') child nodes. Among them, it takes special case for `Gather_Acc`, to which it applies rule 2 that injects a statement to accumulate `msg`.track into `msg_acc`.track.

(b) Rule 4: vectorizing `Scatter` 'Signal'. `Codegen` interprets 'Signal' by updating track in the `msg` with ℓ and signaling neighbors under the condition that at least one of the sources visits the current vertex (via **if** $\ell \neq$ **0 then** condition in rule 4). Note that ℓ could be *'T'* or new node (variable) names introduced by annotation rule 7 of Table 2.

(c) Rules 5-7: track-*guarded vectorization.* `Codegen` interprets compound statements sub-trees of AST via rules 5-7. It vectorizes statements by replacing logical and arithmetical operations over vertex properties with their SIMD version masked by vectors derived from annotations of the AST nodes, such that the vectorized operation only takes effect for sources of which the mask bit values are *True*. The annotation ℓ can only be *'T'* or a distinct variable I introduce by annotation rule 7. `Codegen` interprets *'T'* as constant vector **1** in `Gather` and `Gather_Acc`, and as `msg_acc`.track in

Table 3. Code-generation rules: LHS *denotes an annotated AST node;* RHS *is the generated code.*

Rule 1 Structure Definition (using `msg` as example)		
`Codegen ( #define msg {…` `  <bool> track} )`	⇒	`#define msg {…` `  bitvec<bool> track }`
Rule 2 `Gather_Acc` Function Definition		

`Codegen (`
`  Gather_Acc(`*args* `…)`
`  mark(`T`)` *stmt*
`)`
	⇒	`Gather_Acc(`*args* `…)`
		`  msg_acc.track ← simd_Or(`
		`  msg_acc.track, msg.track)`
		`  Codegen(` mark(T) *stmt* `)`

Rule 3 `Gather`, `Apply`, `Scatter` Function Definition		

`Codegen (`
`  Function Name (`*args* `…)`
`  mark(`T`)` *stmt* `)`
	⇒	Function Name (*args* …)
		`  Codegen(` mark(T) *stmt* `)`

Rule 4 Signal Function Call		

`Codegen (`
`  mark(`ℓ`) Signal(msg))`
// ℓ could be 'T' or new variable name from rule 7
	⇒	**if** $\ell \neq$ **0 then** // NOT `simd_All_Zero` (ℓ)
		`  msg.track ← simd_Set (msg.track,` ℓ`)`
		`  Signal(msg)`

Rule 5 Assignment Statement		

`Codegen (` mark(ℓ) *var* ← *e* `)`
`Codegen (` mark(ℓ) *var* ← `Op(`e_1`) )`
`Codegen (` mark(ℓ) *var* ← e_1 `Op` e_2 `)`
	⇒	*var* ← `simd_mask_Set(`ℓ, *var*, *e*`)`
		var ← `simd_mask_Op(`ℓ, e_1`)`
		var ← `simd_mask_Op(`ℓ, e_1, e_2`)`

Rule 6 Compound Statement		

`Codegen (`
`  mark(`ℓ`)` $stmt_1$ …
`  mark(`ℓ`)` $stmt_n$ `)`
	⇒	{ `Codegen(` mark(ℓ) $stmt_1$ `)`
		  …
		  `Codegen(` mark(ℓ) $stmt_n$ `) }`

Rule 7 IF Selection Statement		

`Codegen (`
`  `$I_t \leftarrow \ell \wedge E$
`  if` I_t `then mark(`I_t`)` $stmt_t$
`  `$I_f \leftarrow \ell \wedge \neg I_t$
`  else mark(`I_f`)` $stmt_f$ `)`
	⇒	`Codegen(` $I_t \leftarrow \ell \wedge E$ `)`
		`Codegen(` mark(I_t) $stmt_t$ `)`
		`Codegen(` $I_f \leftarrow \ell \wedge \neg I_t$ `)`
		`Codegen(` mark(I_f) $stmt_f$ `)`

`Apply` and `Scatter`. When ℓ is a distinct variable I, it uses I directly as a bitvec SIMD mask. When the mask bitvec is literally **1**, `Codegen` simply uses standard SIMD operation without mask.

When AutoMI converts a single-instance GAS **A** into multi-instance $\mathbf{A}_m$, the rules together ensure that $\mathbf{A}_m$ always correctly deduces the exact sources visiting v via `track`, eliminating any incorrect update of streamlined vectorization, *e.g.,* as shown in Example 1. Indeed, (i) `msg.track` of $\mathbf{A}_m$ is always assigned with `bitvec` ℓ that indicates which sources signal neighbors of v in the previous round. (ii) `msg_acc.track` ensures that v is activated by `Scatter` of $\mathbf{A}_m$ only if there exists source s_i such that **A** starting from s_i would signal v in the same round as $\mathbf{A}_m$ does, and `msg_acc.track`$[i]$ is *True* at v.

Example 3: We show how AutoMI generates Algorithm 2 (with the shadowed part) from annotated AST of BFS (Fig. 2). AutoMI first generates structure definition (line 1-2) via rule 1. It then synthesizes function body by invoking `Codegen` at child nodes inductively. For the `Scatter` sub-tree in Fig. 2b, AutoMI uses rule 3 to produce the corresponding `Scatter` function code, with function header (line 14, inherited from 'params' subtree) and function body (line 15-19, derived via rule 6 from 'compound stmt' subtree) generated inductively by following the structure of the annotated AST. □

ALGORITHM 3: Multi-instance Boolean BFS (TrackFree)

```
1  #define vertex_data { bitvec <bool> ans } ;
2  #define msg { bitvec <bool> ans } ;
3  Function Gather(vertex u, edge (u, v), vertex v)
4      return msg ;
5  Function Gather_Acc(vertex v, msg_acc, msg)
6      msg_acc.ans ← simd_Or (msg_acc.ans, msg.ans) ;
7      return msg_acc ;
8  Function Apply(vertex v, msg_acc)
9      changed ← simd_AndNot (v.ans, msg_acc.ans) ;      // bitwise NOT of v.ans and then bitwise AND with msg_acc.ans
10     v.ans ← simd_Or (v.ans, msg_acc.ans) ;
11 Function Scatter(vertex v)
12     if NOT simd_All_Zero (changed) then                // if changed is True for some source
13         msg.ans ← simd_Set (msg.ans, v.ans) ;
14         Signal (msg) ;
```

Theorem 1: *Given a single-instance GAS program* **A** , AutoMI *always generates* $\mathbf{A}_m$ *that is correct as a multi-instance version of* **A**. □

Proof sketch: We prove that, in each round for each vertex v, (a) $\mathbf{A}_m$ correctly deduces the sources visiting v via track, and (b) $\mathbf{A}_m$ correctly updates vertex properties of v for each of these sources.

Here (a) follows directly from the property of code-generation rules. We prove (b) by showing, via induction on the program structure of **A**, that when a rule of Table 3 is employed for the induction step, it correctly synthesizes code generated for sub-structures of **A**. For brevity, we show that rule 7 of Table 3 is correct; other rules are similar. Suppose annotation label ℓ indicates sources that are visiting vertex v. Observe that $stmt_t$ branch is executed by $\mathbf{A}_m$ masked by I_t, which is effective only for sources that are visiting v and satisfies the if condition (E); similarly for $stmt_f$ branch. As a result, $\mathbf{A}_m$ synthesized by rule 7 is correct as long as $stmt_t$ and $stmt_f$ are generated correctly, which is confirmed by the induction hypothesis. □

Remark. Theorem 1 assumes that GAS functions do not contain loops. That said, one can extend its coverage by adding rules in AutoMI to support other control flow statements *e.g.,* for and while loops, while still assuring the correctness of Theorem 1. We remark that for and while loops are rarely used in GAS programs, which are essentially local functions running concurrently on each vertex.

4 TRACK-FREE VECTORIZATION

We have seen that track-guarded vectorization warrants the correctness of AutoMI for populating GAS algorithms. In this section, we show that there exist GAS programs, referred to as TrackFree *algorithms*, that can be converted using streamlined vectorization, without track. Below we describe TrackFree and see how it fits into AutoMI to populate significantly simpler and faster GAS programs.

Example 4: Consider a variant of Algorithm 1 for *Boolean* BFS which returns, for each vertex v, if v is reachable from the source via a BFS traversal. It maintains only a boolean vertex property ans such that v.ans is *True* if v is reachable; initially, v.ans is *True* if v is the source and is *False* otherwise. At each vertex v, Gather pulls ans of all inward neighbors and Gather_Acc accumulates ans into msg_acc via logical $\vee$, such that msg_acc is *True* if ans of one of its neighbors is *True*. Apply then updates v.ans with v.ans $\vee$ msg_acc, and records in changed if v.ans is changed: it sets

changed to *True* if msg_acc.ans $\wedge \neg v$.ans is *True*. Finally, Scatter signals outward neighbors of v if changed is *True*, with v.ans as the signal msg.

A direct vectorization of Boolean BFS would give us Algorithm 3 (vectorized operations are colored in blue). One can verify that Algorithm 3 is a correct multi-instance Boolean BFS algorithm, and is simpler than the one that AutoMI would generate using track. □

TrackFree algorithms. As illustrated in Example 4, there are GAS algorithms that can be populated into their multi-instance version without the need of deducing and injecting track as the mask of vectorized (SIMD) operations on vertex properties.

Consider GAS algorithm **A**. Let $\mathbf{A}_m$ be the multi-instance program generated by AutoMI as in Section 3.2, using all rules of Tables 2 and 3. Let $\mathbf{A}_m^{\mathsf{TF}}$ be the version deduced by AutoMI with

- (annotation phase) rule 1 of Table 2 disabled; and
- (code generation phase) rule 1 of Table 3 disabled, steps containing track (line 2 on RHS) of rules 2 and 4 removed, and *'T'* replaced with **1** in all rules of Table 3.

Intuitively, when AutoMI operates with these changes, it generates $\mathbf{A}_m^{\mathsf{TF}}$ as streamlined vectorized version of **A**, without using track. Algorithm **A** is TrackFree if $\mathbf{A}_m$ and $\mathbf{A}_m^{\mathsf{TF}}$ return the same answer over the same input: given any graph G and set S of sources vertices, the answer computed by $\mathbf{A}_m$ for each source $s \in S$ is the same as that by $\mathbf{A}_m^{\mathsf{TF}}$. That is, $\mathbf{A}_m^{\mathsf{TF}}$ is also a multi-instance version of **A**.

Benefits. When **A** is TrackFree, AutoMI generates $\mathbf{A}_m^{\mathsf{TF}}$ instead of $\mathbf{A}_m$ as the multi-instance version of **A**. Compared to $\mathbf{A}_m$, the TrackFree version $\mathbf{A}_m^{\mathsf{TF}}$ has two main advantages:

Simpler code. $\mathbf{A}_m^{\mathsf{TF}}$ is much simpler than $\mathbf{A}_m$ as it is streamlined vectorization of **A** with each operation over vertex properties replaced with their vectorized version without track, as if all sources visit v when GAS functions fire at v. For instance, Algorithm 3 is exactly the TrackFree multi-instance Boolean BFS that AutoMI would generate with the above optimization. Comparing to Algorithm 2 and for Integer BFS (Algorithm 1), the changes highlighted in Algorithm 3 are much more regular.

Higher performance. $\mathbf{A}_m^{\mathsf{TF}}$ is more efficient than $\mathbf{A}_m$ due to (a) better exploitation of vectorization, *i.e.,* SIMD without mask, and (b) no need of deducing track, which can be a significant overhead in particular for GAS algorithms with lightweight Apply functions. As we will see in Section 7, TrackFree multi-instance Boolean BFS improves Boolean BFS over PowerLyra [20], a popular GAS system, by 14.82 to 27.83 times while the speedup with the standard multi-instance version without TrackFree is 6.71 to 13.64 times. That is, TrackFree improves multi-instance Boolean BFS by over 2 times.

5 AN ALGEBRAIC CHARACTERIZATION

While Section 4 has shown that TrackFree is effective in improving the performance of multi-instance GAS algorithms and greatly simplifying their programs, it does not tell us when TrackFree can apply. To address this, we develop a formal characterization below.

We first show how to recast a GAS algorithm **A** into an algebra expression $e_{\mathbf{A}}$ (Section 5.1). We then reduce the decision of TrackFree to checking if $e_{\mathbf{A}}$ has a form of *idempotence* (Section 5.2).

5.1 An Algebra for GAS Algorithms

Algebra. To assist with the decision of TrackFree, we cast GAS in an extension of linear algebra.

Syntax. We consider algebra expressions defined by grammar:

$$e[\mathbf{x}_{\mathsf{ans}}] ::= V \mid e^T \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid f(e_1, \ldots, e_i) \mid e^k[\mathbf{x}_1, \ldots, \mathbf{x}_j](e_1, \ldots e_j)$$

where $V, \mathbf{x} \in \mathcal{V}$ are matrix variables. Expression e builds one or multiple matrices from the variables via matrix transpose (e^T), matrix addition (e_1+e_2), matrix multiplication ($e_1 \cdot e_2$), pointwise application

of function f to elements of $e_1, \ldots, e_i$, and iteration e^k that applies e to matrices $e_1, \ldots, e_i$ repeatedly for k times (k can also be ∞). Here pointwise function $f : \mathbb{N}^i \to \mathbb{N}$ uses addition $\oplus$ and multiplication $\otimes$ of a semiring of choice to combine elements of i matrices. For graph algorithms, we only need to consider square matrices and vectors: each variable in $\mathcal{V}$ is either a square matrix of type (n, n), a vector of type $(n, 1)$ or a scalar of type $(1, 1)$, where n is the number of vertices of the graph.

We consider well-typed expressions. For example, for matrix-vector multiplication $V \cdot \mathbf{x}$, V is of type (n, n) and $\mathbf{x}$ has type $(n, 1)$. For pointwise function $f(e_1, \ldots, e_i)$ to make sense, $e_1, \ldots, e_i$ must have the same type, *e.g.,* all are (n, n) matrices or $(n, 1)$ vectors.

For $e^k[\mathbf{x}_1, \ldots, \mathbf{x}_j](e_1, \ldots, e_j)$, each $e_i (i \in [1, j])$ is defined over the j variables $\mathbf{x}_1, \ldots, \mathbf{x}_j$; moreover, the type of (the output of) e_i is the same as that of $\mathbf{x}_i$. We restrict that $e_1, \ldots, e_j$ do not contain e^k as a sub-expression to avoid unnecessary recursion for GAS programs.

Semantics. Consider a valuation function ν that maps each matrix variable to a matrix or vector of appropriate type. For GAS algorithms, this will be either the adjacent matrix A of the graph or the default values of vertex properties set during initialization. For a well-typed e, the matrix $\nu(e)$ obtained by evaluating e with ν is constructed by following the semantics of each operation of e.

Of particular interest are $e_f := f(e_1, \ldots, e_i)$ and $e^k[\mathbf{x}_1, \ldots, \mathbf{x}_j](e_1, \ldots, e_j)$. Specifically, $\nu(e_f)$ returns a matrix or vector of the same type as e_1, with values computed through elementwise operations specified by f. Further, $\nu(e^k)$ allows iterative update on the j matrices computed by e_1, …, e_j for k rounds; in round r, it takes the output of e_1, …, e_j in round $r - 1$ as values for $\mathbf{x}_1$, …, $\mathbf{x}_j$, respectively, and evaluate e (*i.e.,* $e_1, \ldots, e_j$) with them. Note that each e_i may use multiple or even all variables $\mathbf{x}_1$, …, $\mathbf{x}_j$. In particular, e^∞ iterates as such until the output of e_1, …, e_j stays unchanged.

The result of $e[\mathbf{x}_{\text{ans}}]$ is the value of $\mathbf{x}_{\text{ans}}$ when evaluating e with ν.

Remark. We remark that the extended algebra resembles for-MATLANG [23] and the widely studied semiring frameworks in formality to express the vertex property update logic in GAS programs. However, it serves a very different purpose. As will be shown below, by extending linear algebra with "restricted recursion" ($e^k[\mathbf{x}_1, \ldots, \mathbf{x}_j](e_1, \ldots, e_j)$), we can explicitly express the iterative execution of GAS programs, which provides a means for characterizing their `TrackFree` property. This is in contrast to existing semiring frameworks that model and generalize query semantics. For instance, Mohri [42] uses semirings to generalize the distance function of shrotest-distance queries, and Khamis et al. [13] generalizes datalog to Datalog° with partially ordered pre-semiring structures.

Recasting GAS programs. Given a single instance GAS algorithm **A**, we derive the following:

❶ $e_{\text{A}}[\mathbf{x}_{\text{ans}}] := e^\infty[\mathbf{x}_1, \ldots, \mathbf{x}_p]$ *// p vertex properties, encoded by $\mathbf{x}_1, \ldots, \mathbf{x}_p$*
❷ $e[\mathbf{x}_1, \ldots, \mathbf{x}_p] := (e_1, \ldots, e_p)$ *// e_i: update logic of property $\mathbf{x}_i$ in a round*
❸ δ_{A} := signal logic in the `Scatter` function

Here part ❶ is to declare matrix (vector) variables for all the vertex properties defined in the preamble of **A**. The critical step is part ❷, which recasts the `Apply` function logic in the algebra. In particular, each e_i expresses how `Apply` updates the vertex property encoded by $\mathbf{x}_i$ in each round. To simplify the process, we account for active vertices only and derive e_i by assuming all vertices are active.

Part ❸ captures how **A** activates vertices in `Scatter`. Specifically, δ_{A} expresses the condition under which an active vertex would signal its neighbors for the next round. It returns a boolean vector such that $\delta_{\text{A}}[v]$ is *True* if v should signal its neighbors in `Scatter` and is *False* otherwise.

Example 5: [Recasting Integer BFS] We show how to recast Algorithm 1 for BFS. Following its preamble, we use $\mathbf{x}_{\text{ans}}$ and $\mathbf{x}_{\text{visited}}$ to encode vertex properties ans and visited, respectively. Part ❶ is:

$$e_{\text{iBFS}}[\mathbf{x}_{\text{ans}}] := e^k[\mathbf{x}_{\text{ans}}, \mathbf{x}_{\text{visited}}](e_{\text{ans}}, e_{\text{visited}}).$$

We next derive e from `Apply`. For an active vertex v, v.ans ($\mathbf{x}_{\mathsf{ans}}[v]$) value is unchanged if v.visited ($\mathbf{x}_{\mathsf{visited}}[v]$) is *True*, otherwise it is set to the aggregated message `msg_acc` at v (lines 9-10, Algorithm 1). Hence, the value of $\mathbf{x}_{\mathsf{ans}}[v]$ after executing `Apply` once can be written as $\mathbf{x}_{\mathsf{visited}}[v]\mathbf{x}_{\mathsf{ans}}[v] + (1 - \mathbf{x}_{\mathsf{visited}}[v])\mathbf{msg\_acc}[v]$. Treating all vertices as active, one can simply "vectorize" this into $e_{\mathsf{ans}}[\mathbf{x}_{\mathsf{ans}}] := f(\mathbf{x}_{\mathsf{visited}}, \mathbf{x}_{\mathsf{ans}}, \mathbf{1} - \mathbf{x}_{\mathsf{visited}}, \mathbf{msg\_acc})$, where f is pointwise function:

$$f(x, y, z, w) = (x \wedge y) \vee (z \wedge w).$$

Further, from lines 6 and 15, we see that `msg_acc` of all vertices is computed by $A^T \cdot \mathbf{x}_{\mathsf{ans}}$, where A is the adjacent matrix of graph G and $A^T \cdot \mathbf{x}_{\mathsf{ans}}$ uses tropical semiring with `Min` for $\oplus$ (line 6) and + for $\otimes$ (line 15). Similarly, we can derive e_{visited}. This gives us part ❷:

$$\begin{array}{rcl} e[\mathbf{x}_{\mathsf{ans}}, \mathbf{x}_{\mathsf{visited}}] & := & (f(\mathbf{x}_{\mathsf{visited}}, \mathbf{x}_{\mathsf{ans}}, \mathbf{1} - \mathbf{x}_{\mathsf{visited}}, \mathbf{msg\_acc}), \mathbf{1}) \\ \mathbf{msg\_acc} & := & A^T \cdot \mathbf{x}_{\mathsf{ans}} \end{array}$$

Here $\mathbf{0}$ (resp. $\mathbf{1}$) is a boolean vector of all *False* (resp. *True*).

For part ❸, observe that lines 14, 16 show that vertex v signals its neighbors if `changed` is *True*. Thus, we have $\delta_{\mathsf{iBFS}} := \mathbf{changed}$. □

5.2 Deciding `TrackFree` via Algebra Idempotence

We next develop a characterization of `TrackFree` by analyzing the algebraic representation $e_{\mathbf{A}}$ of the GAS programs **A** of Section 5.1. Below we first present our main result, two *semantic idempotence* properties of $e_{\mathbf{A}}$ for `TrackFree` to hold for **A**. We then prove its correctness by connecting idempotence with algebra equivalence, a semantic characterization that is further reduced to `TrackFree`.

Semantic idempotence. We define two idempotent properties.

Active idempotence. Let $\lambda_{\mathtt{msg\_acc}}$ be the value of accumulated message. *Active idempotence* specifies that repeatedly applying the `Apply` function with the *same* accumulated message value would not change the value of vertex properties. That is:

$$e[\mathbf{x}_1, \ldots, \mathbf{x}_p]\langle\lambda_{\mathtt{msg\_acc}}\rangle = e[e[\mathbf{x}_1, \ldots, \mathbf{x}_p]\langle\lambda_{\mathtt{msg\_acc}}\rangle]\langle\lambda_{\mathtt{msg\_acc}}\rangle$$

Here $e[\mathbf{x}_1, \ldots, \mathbf{x}_p]\langle\lambda_{\mathtt{msg\_acc}}\rangle$ is to explicitly specify that `Apply` uses $\lambda_{\mathtt{msg\_acc}}$ as the accumulated message **msg_acc**. As shown in Example 5, $\lambda_{\mathtt{msg\_acc}}$ is deduced from vertex properties in the previous round, *e.g.,* $A^T \cdot \mathbf{x}_{\mathsf{ans}}$ for BFS. To show that the repeated application of e uses the same `msg_acc`, we add $\lambda_{\mathtt{msg\_acc}}$ as a parameter to e.

Inactive idempotence. Let $\alpha_{\mathtt{msg\_acc}}$ be the `msg_acc` value of vertices that have no source vertices as their neighbors. That is, $\alpha_{\mathtt{msg\_acc}}$ is the `msg_acc` value determined by the initialized vertex properties of non-source vertices. *Inactive idempotence* requires that applying the `Apply` logic with $\alpha_{\mathtt{msg\_acc}}$ does not change vertex properties. Putting in algebraic expression, this is expressed as:

$$[\mathbf{x}_1, \ldots, \mathbf{x}_p] = e[\mathbf{x}_1, \ldots, \mathbf{x}_p]\langle\alpha_{\mathtt{msg\_acc}}\rangle$$

Intuitively, active idempotence assures that repeated invocations of `Apply` with the same `msg_acc` value would not make a difference. Inactive idempotence ensures that invoking `Apply` on inactive vertices would not alter vertex properties either. When the properties holds for $e_{\mathbf{A}}$, we say that $e_{\mathbf{A}}$ is both *active* and *inactive idempotent*.

Recall that the signal logic in the `Scatter` function is encoded by $\delta_{\mathbf{A}}$. We say that $\delta_{\mathbf{A}}$ is *regular* if $\delta_{\mathbf{A}}$ is either (a) $\mathbf{1}$, *i.e.,* always signal neighbors of active vertices, or (b) **changed**, *i.e.,* signal the neighbors of v only if vertex properties of v are updated in the current round (*i.e.,* `changed` of v is *True*). Then we have the following.

Theorem 2: *For any GAS algorithm* **A**, *if (a)* $e_{\mathbf{A}}$ *is both active and inactive idempotent, and (b)* $\delta_{\mathbf{A}}$ *is regular, then* **A** *is* `TrackFree`. □

Example 6: Continuing Example 5. First observe that δ_{iBFS} in part ❸ is regular as it satisfies condition (b) above. However, part ❷ e of e_{iBFS} is active idempotent, but not inactive idempotent. To see e_{iBFS} is active idempotent, observe that for any $\lambda_{\texttt{msg\_acc}}$, we have $e[e[\mathbf{x}_{\text{ans}}, \mathbf{x}_{\text{visited}}]\langle\lambda_{\texttt{msg\_acc}}\rangle]\langle\lambda_{\texttt{msg\_acc}}\rangle = e[\mathbf{x}_{\text{ans}}, \mathbf{x}_{\text{visited}}]\langle\lambda_{\texttt{msg\_acc}}\rangle$ since the first application of e sets $\mathbf{x}_{\text{visited}}$ to *True* and prevents any repeated applications from changing vertex properties $\mathbf{x}_{\text{ans}}$ and $\mathbf{x}_{\text{visited}}$. To see e_{iBFS} is not inactive idempotent, observe that $e[\mathbf{x}_{\text{ans}}, \mathbf{x}_{\text{visited}}]\langle\alpha_{\texttt{msg\_acc}}\rangle = (\mathbf{x}_{\text{visited}} * \mathbf{x}_{\text{ans}} + (\mathbf{1} - \mathbf{x}_{\text{visited}}) * \alpha_{\texttt{msg\_acc}}, \mathbf{1})$, regardless of what $\alpha_{\texttt{msg\_acc}}$ is used, e always sets $\mathbf{x}_{\text{visited}}$ to *True* which is however *False* initially for non-source vertices. This is consistent with Section 3 that Algorithm 1 is not `TrackFree`.

[Boolean BFS] Recall Boolean BFS in Example 4. Recast in algebra:

$$\begin{aligned} e_{\text{bBFS}}[\mathbf{x}_{\text{ans}}] &:= e^{\infty}_{\text{ans}}[\mathbf{x}_{\text{ans}}] \\ e_{\text{ans}}[\mathbf{x}_{\text{ans}}] &:= \mathbf{x}_{\text{ans}} \oplus \mathbf{msg\_acc} \\ \mathbf{msg\_acc} &:= A^T \cdot \mathbf{x}_{\text{ans}} \\ \delta_{\text{bBFS}} &:= \mathbf{changed} \end{aligned}$$

Here $\oplus$ is logical or ($\vee$) and $A^T \cdot \mathbf{x}_{\text{ans}}$ uses boolean semiring.

To see e_{bBFS} is active idempotent, observe that for any $\lambda_{\texttt{msg\_acc}}$, we have $e_{\text{ans}}[e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\texttt{msg\_acc}}\rangle]\langle\lambda_{\texttt{msg\_acc}}\rangle = (\mathbf{x}_{\text{ans}} \vee \lambda_{\texttt{msg\_acc}}) \vee \lambda_{\texttt{msg\_acc}} = \mathbf{x}_{\text{ans}} \vee \lambda_{\texttt{msg\_acc}} = e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\texttt{msg\_acc}}\rangle$. To see e_{bBFS} is inactive idempotent, note that the initialized `msg_acc` value $\alpha_{\texttt{msg\_acc}}$ is *False*. As a result, we have that $e_{\text{bBFS}}[\mathbf{x}_{\text{ans}}]\langle\alpha_{\texttt{msg\_acc}}\rangle = \mathbf{x}_{\text{ans}} \vee \mathit{False} = \mathbf{x}_{\text{ans}}$. That is, e_{bBFS} is also inactive idempotent. Note that δ_{bBFS} is regular, by Theorem 2 we know that Boolean BFS is `TrackFree`.

This confirms Example 4 that Algorithm 3 is indeed a correct multi-instance Boolean BFS, deduced by AutoMI with `TrackFree`. There are also cases where an algorithm is inactive idempotent but not active idempotent. As an example, if we set $\oplus$ to logical XOR in $e_{\text{ans}}[\mathbf{x}_{\text{ans}}] := \mathbf{x}_{\text{ans}} \oplus \mathbf{msg\_acc}$ of Boolean BFS expression above, then it becomes inactive idempotent but not active idempotent, contrary to Integer BFS in Example 6. Indeed, it is inactive idempotent because $e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\alpha_{\texttt{msg\_acc}}\rangle = \mathbf{x}_{\text{ans}}$ XOR $\mathit{False} = \mathbf{x}_{\text{ans}}$. However, it is not active idempotent: note that if $\lambda_{\texttt{msg\_acc}}$ is *True* and $\mathbf{x}_{\text{ans}}$ is $\mathbf{0}$, we have $e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\texttt{msg\_acc}}\rangle = \mathbf{1}$, meanwhile $e_{\text{ans}}[e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\texttt{msg\_acc}}\rangle]\langle\lambda_{\texttt{msg\_acc}}\rangle = \mathbf{0}$. □

Proof of Theorem 2. We prove Theorem 2 in two steps. (1) We first develop a characterization of `TrackFree` by bridging `TrackFree` of $\mathbf{A}$ and the equivalence between $e_{\mathbf{A}}$ and a expression $e'_{\mathbf{A}}$ derived from $e_{\mathbf{A}}$. (2) We then show that conditions of Theorem 2 warrant equivalence between $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$.

Step 1: characterization of `TrackFree`. Recall that when we recast $\mathbf{A}$ into the $e_{\mathbf{A}}$ in Section 5.1, we treat all vertices as active to focus on only the `Apply` logic that updates each vertex property, while using $\delta_{\mathbf{A}}$ to abstract the signal logic. We next construct an expression $e'_{\mathbf{A}}$ that combines $e_{\mathbf{A}}$ and $\delta_{\mathbf{A}}$ so as to precisely express $\mathbf{A}$. The idea is to add a condition to each sub-expression that updates a vertex property, *e.g.,* $\mathbf{x}_{\text{ans}} \oplus \mathbf{msg\_acc}$ for $\mathbf{x}_{\text{ans}}$ in Example 6 for Boolean BFS. The condition, encoded by a new expression $\text{act}(\delta_{\mathbf{A}}, \mathbf{t})$, is defined as $g(A^T \cdot \delta_{\mathbf{A}}, A^T \cdot \mathbf{t})$, where g is a pointwise function $g(x, y) = x \wedge y$ and $\mathbf{t}$ is a boolean vector such that $\mathbf{t}[v]$ is *True* if vertex v is source and is *False* otherwise. We then use act to condition each sub-expression $e_{\mathbf{x}}$ of $e_{\mathbf{A}}$ that updates vertex property $\mathbf{x}$, such that $e_{\mathbf{A}}$ only updates $\mathbf{x}$ for vertices that are active by $\mathbf{A}$. It does this by replacing $e_{\mathbf{x}}$ with $f(1 - \text{act}(\delta_{\mathbf{A}}, \mathbf{t}), \mathbf{x}, \text{act}(\delta_{\mathbf{A}}, \mathbf{t}), e_{\mathbf{x}})$ (recall function f from Example 5).

Example 7: Recall e_{bBFS} for Boolean BFS in Example 6. We derive e'_{bBFS} from e_{bBFS} (changes in blue):

$$\begin{aligned} e'_{\text{bBFS}}[\mathbf{x}_{\text{ans}}] &:= (e'_{\text{ans}})^{\infty}[\mathbf{x}_{\text{ans}}, \mathbf{t}] \\ e'_{\text{ans}}[\mathbf{x}_{\text{ans}}, \mathbf{t}] &:= (f(1 - \text{act}(\delta_{\text{bBFS}}, \mathbf{t}), \mathbf{x}_{\text{ans}}, \text{act}(\delta_{\text{bBFS}}, \mathbf{t}), \mathbf{x}_{\text{ans}} \oplus \mathbf{msg\_acc}), \text{act}(\delta_{\text{bBFS}}, \mathbf{t})) \\ \mathbf{msg\_acc} &:= A^T \cdot \mathbf{x}_{\text{ans}} \\ \delta_{\text{bBFS}} &:= \mathbf{changed} \\ \text{act}(\delta_{\text{bBFS}}, \mathbf{t}) &:= g(A^T \cdot \mathbf{t}, A^T \cdot \delta_{\text{bBFS}}) \end{aligned}$$

□

Consider GAS algorithm $\mathbf{A}$ and its recast algebra expression $e_{\mathbf{A}} := e^{\infty}$. Let $e'_{\mathbf{A}} := (e')^{\infty}$ be the expression derived from $e_{\mathbf{A}}$ as described above. We say that $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$ are *equivalent*, denoted by $e_{\mathbf{A}} \equiv e'_{\mathbf{A}}$, if for any graph G (its adjacent matrix A), the evaluation results of $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$ over A are the same. The case for $e_{\mathbf{A}} := e^k$ for some number k is similar. Then we have the following.

Lemma 3: *For any GAS algorithm* $\mathbf{A}$*, if* $e_{\mathbf{A}} \equiv e'_{\mathbf{A}}$ *then* $\mathbf{A}$ *is* `TrackFree`. □

Proof sketch: To prove Lemma 3, we first present the following result. Let $\mathbf{A}_{\mathsf{all}}$ be another GAS algorithm that is identical to $\mathbf{A}$ except that $\mathbf{A}_{\mathsf{all}}$ signals all vertices in `Scatter` for each round. Then:

Proposition 4: *For any GAS algorithm* $\mathbf{A}$*,* $\mathbf{A}_{\mathsf{all}}$ *is* `TrackFree`. □

We then show that $e'_{\mathbf{A}}$ equivalently expresses $\mathbf{A}$. Indeed, first observe that $\mathbf{t}$ updated by $\mathsf{act}(\delta_{\mathbf{A}}, \mathbf{t})$ exactly identifies which vertices are active in each round: $\mathsf{act}(\delta_{\mathbf{A}}, \mathbf{t})$ returns *True* for a vertex v iff there exists a neighbor u that (i) is active in the previous round (via $A^T \cdot \mathbf{t}$) and (ii) signals v in `Scatter` (via $A^T \cdot \delta_{\mathbf{A}}$). Hence, by its construction, $e'_{\mathbf{A}}$ exactly captures $\mathbf{A}$. Further observe that $e_{\mathbf{A}}$ equivalently expresses $\mathbf{A}_{\mathsf{all}}$ as it follows the vertex update logic of $\mathbf{A}$ but treats all vertices as active. Hence, if $e_{\mathbf{A}} \equiv e'_{\mathbf{A}}$ then $\mathbf{A}$ is equivalent to $\mathbf{A}_{\mathsf{all}}$. By Proposition 4, $\mathbf{A}$ must also be `TrackFree`. □

Step 2: idempotence ⇒ equivalence. While Lemma 3 tells us that `TrackFree` of $\mathbf{A}$ can be decided by analyzing the equivalence of $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$, the latter, however, is not straightforward either. To this end, we show that the condition in Theorem 2 suffices to warrant the equivalence of $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$.

Lemma 5: *For any GAS algorithm* $\mathbf{A}$*, if* $\delta_{\mathbf{A}}$ *is regular and* $e_{\mathbf{A}}$ *is both active and inactive idempotent, then* $e_{\mathbf{A}} \equiv e'_{\mathbf{A}}$ *.* □

Proof sketch. We prove a stronger result: when $\delta_{\mathbf{A}}$ is regular and $e_{\mathbf{A}}$ is active and inactive idempotent, the values of vertex properties computed by $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$ in *each* round are identical. We show this by induction on the total number of rounds that $e_{\mathbf{A}}$ takes to converge. The crux is to show that $e_{\mathbf{A}}$ and $e'_{\mathbf{A}}$ make the same changes to all vertex properties when the $\delta_{\mathbf{A}}$ is regular and $e_{\mathbf{A}}$ is idempotent. □

Putting Lemma 3 and Lemma 5 together, we complete the proof of Theorem 2. As will be shown shortly in Section 6, Theorem 2 enables us to syntactically check if our single-instance GAS algorithm can be automatically populated with `TrackFree` enabled.

Remark. Theorem 2 provides a sufficient condition for `TrackFree`; however, it is not necessary. As an example, consider a variant $\mathbf{A}$ of Boolean BFS (Algorithm 3) that additionally maintains visited of Integer BFS (Algorithm 1) as a "dummy" vertex property that is not used elsewhere. In algebra,

$$\begin{aligned} e_{\mathbf{A}}[\mathbf{x}_{\mathsf{ans}}] &:= e^{\infty}[\mathbf{x}_{\mathsf{ans}}, \mathbf{x}_{\mathsf{visited}}](e_{\mathsf{ans}}, e_{\mathsf{visited}}) \\ e[\mathbf{x}_{\mathsf{ans}}, \mathbf{x}_{\mathsf{visited}}] &:= (\mathbf{x}_{\mathsf{ans}} \oplus \mathtt{msg\_acc}, \mathbf{1}) \\ \mathtt{msg\_acc} &:= A^T \cdot \mathbf{x}_{\mathsf{ans}} \\ \delta_{\mathbf{A}} &:= \mathbf{changed} \end{aligned}$$

Intuitively, $e_{\mathbf{A}}$ inherits e_{bBFS} from Boolean BFS (Example 6) but includes $\mathbf{x}_{\mathsf{visited}}$ from Integer BFS (Example 5). Note that $\mathbf{A}$ is `TrackFree` as $\mathbf{x}_{\mathsf{ans}}$ in $e_{\mathbf{A}}$ effectively resembles $\mathbf{x}_{\mathsf{ans}}$ in e_{bBFS}, independent of $\mathbf{x}_{\mathsf{visited}}$. However, due to $\mathbf{x}_{\mathsf{visited}}$, $e_{\mathbf{A}}$ is not inactive idempotent for the same reason that e_{iBFS} is not.

6 APPLICATIONS

In this section, we demonstrate how AutoMI generates multi-instance version of common graph algorithms. We have already seen Integer BFS and Boolean BFS; below we focus on shortest path (SSSP), greedy graph coloring, and advanced graph analytical algorithms.

SSSP. Single-source shortest path (SSSP) returns, for each vertex v, the shortest distance from the source vertex to v. AutoMI generates multi-instance SSSP from the GAS implementation of the

ALGORITHM 4: AutoMI generated multi-instance Bellman-Ford (SSSP)

```
1  #define vertex_data { array <int> ans } ;
2  #define msg { array <int> ans } ;
3  Function Gather(vertex u, edge (u, v), vertex v)
4      msg.ans ← simd_Add (msg.ans, w(u, v)) ;
5      return msg ;
6  Function Gather_Acc(vertex v, msg_acc, msg)
7      msg_acc.ans ← simd_Min (msg_acc.ans, msg.ans) ;
8      return msg_acc;
9  Function Apply(vertex v, msg_acc)
10     mask ← simd_CmpGt (v.ans, msg_acc.ans) ;
11     v.ans ← simd_mask_Set (mask, v.ans, msg_acc.ans) ;
12     changed ← simd_mask_Set (mask, changed, 1) ;
13 Function Scatter(vertex v)
14     if NOT simd_All_Zero (changed) then        // if changed is True for some source
15         msg.ans ← simd_Set (msg.ans, v.ans) ;
16         Signal (msg) ;                         // activates v's outward neighbors
```

classic Bellman-Ford algorithm [24] with `TrackFree` enabled, as shown in Algorithm 4. It maintains an integer vertex property `ans` that stores the tentative shortest distances of a vertex from all the sources; initially, $v.\texttt{ans}[i]$ is $+\infty$ if v is not source s_i and is 0 otherwise.

For an active vertex v, `Gather` pulls `msg` (`ans`) from each inward neighbor u and increments it with edge weight $w(u, v)$ (line 4); their minimum is then assigned to `msg_acc` by `Gather_Acc`, by aggregating with `Min` (line 7). Then `Apply` checks if v.`ans` is greater than `msg_acc` (line 10); if so, it sets v.`ans` to `msg_acc` (line 11) and marks v.`ans` as changed (line 12). In `Scatter` function, vertex v signals its neighbors with v.`ans` (lines 15-16) only if v.`ans` is updated in the current round (line 14).

We next justify why AutoMI populates SSSP with `TrackFree`. By "devectorizing" operations of Algorithm 4 (in blue), one can readily see the single-instance SSSP in GAS, which recast in algebra is:

$$\begin{aligned} e_{\text{SSSP}}[\mathbf{x}_{\text{ans}}] &:= e^{\infty}_{\text{ans}}[\mathbf{x}_{\text{ans}}] \\ e_{\text{ans}}[\mathbf{x}_{\text{ans}}] &:= \mathbf{x}_{\text{ans}} \oplus \textbf{msg\_acc} \\ \textbf{msg\_acc} &:= A^T \cdot \mathbf{x}_{\text{ans}} \\ \delta_{\text{SSSP}} &:= \textbf{changed} \end{aligned}$$

where A^T is the transposed adjacency matrix of weighted graph G and $A^T \cdot \mathbf{x}_{\text{ans}}$ uses tropical semiring with `Min` for $\oplus$ and + for $\otimes$. SSSP is `TrackFree` since δ_{SSSP} is regular and e_{SSSP} is both active and inactive idempotent. To see e_{SSSP} is active idempotent, observe that for any $\lambda_{\text{msg\_acc}}$, we have $e_{\text{ans}}[e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle]\langle\lambda_{\text{msg\_acc}}\rangle = \texttt{Min}(\texttt{Min}(\mathbf{x}_{\text{ans}}, \lambda_{\text{msg\_acc}}), \lambda_{\text{msg\_acc}}) = \texttt{Min}(\mathbf{x}_{\text{ans}}, \lambda_{\text{msg\_acc}})$, which is equal to $e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle$. To see e_{SSSP} is inactive idempotent, observe that the initialized `msg_acc` value $\alpha_{\text{msg\_acc}}$ is $+\infty$. As a result, $e_{\text{SSSP}}[\mathbf{x}_{\text{ans}}]\langle\alpha_{\text{msg\_acc}}\rangle = \texttt{Min}(\mathbf{x}_{\text{ans}}, +\infty) = \mathbf{x}_{\text{ans}}$. By Theorem 2, SSSP is `TrackFree` and hence Algorithm 4 is correct.

Greedy graph coloring. Given a graph G and a source vertex s with some color c, a graph coloring of G *w.r.t.* s is an assignment of colors to all remaining vertices such that no two adjacent vertices in G share the same color, with the objective to minimize the number of colors used. Given k sources, multi-instance graph coloring is to find k independent graph colorings, each for a source. We consider greedy graph coloring (GC) [24], a vertex-centric algorithm that iteratively assigns minimum color not used by neighbors to active vertices, with the color state encoded as a bitvec

ALGORITHM 5: AutoMI generated multi-instance Greedy Graph Coloring

```
1  #define vertex_data { array <bitvec> ans } ;
2  #define msg { array <bitvec> ans } ;
3  Function Gather(vertex u, edge (u, v), vertex v)
4      return msg ;
5  Function Gather_Acc(vertex v, msg_acc, msg)
6      msg_acc.ans ← simd_Or (msg_acc.ans, msg.ans) ;          // agg neighbors' colors
7      return msg_acc;
8  Function Apply(vertex v, msg_acc)
9      mask1 ← simd_CmpNeq (msg_acc.ans, 0) ;                  // filter neighbors already colored
10     temp ← simd_mask_Add (mask1, msg_acc.ans, 1) ;
11     c ← simd_mask_AndNot (mask1, temp, msg_acc.ans) ;       // min. avail. color
12     mask2 ← simd_mask_CmpNeq (mask1, v.ans, c) ;            // sources for new v's color
13     v.ans ← simd_mask_Set (mask2, v.ans, c) ;               // update v's color
14     changed ← simd_mask_Set (mask2, changed, 1) ;           // update changed
15 Function Scatter(vertex v)
16     if NOT simd_All_Zero (changed) then                     // if changed is True for some source
17         msg.ans ← simd_Set (msg.ans, v.ans) ;               // send out v's color for next rd.
18         Signal (msg) ;                                      // activates v's outward neighbors & send out msg
```

array [22] ans for each v such that the i-th bit of ans$[j]$ is 1 if v is assigned with color i for source s_j (assume colors are indexed by integers starting from 1).

AutoMI generates multi-instance GC (Algorithm 5) from the classic GAS GC [24] with TrackFree. For an active vertex v, Apply checks if the color state aggregated from neighbors indicates no color has been assigned (*i.e.,* **0**, line 9), computes the minimum color **c** available through bitwise operation (lines 10-11), updates the color state v.ans with **c** using masked SIMD (lines 12-14). Note that v.ans is changed only if aggregated msg_acc.ans is not **0** and v.ans is different from **c**. The message pulled by active vertex v from neighbor u in Gather (line 4) carries the color state of u assigned in previous round by Scatter (line 17), aggregated via bitwise OR operation in Gather_Acc (line 6).

To see why AutoMI enables TrackFree, recast GC in algebra as:

$$\begin{aligned} e_{\text{GC}}[\mathbf{x}_{\text{ans}}] &:= e^{\infty}_{\text{ans}}[\mathbf{x}_{\text{ans}}] \\ e_{\text{ans}}[\mathbf{x}_{\text{ans}}] &:= h(\mathbf{x}_{\text{ans}}, \textbf{msg\_acc}) \\ \textbf{msg\_acc} &:= A^T \cdot \mathbf{x}_{\text{ans}} \\ \delta_{\text{GC}} &:= \textbf{changed} \end{aligned}$$

$A^T \cdot \mathbf{x}_{\text{ans}}$ uses bitwise OR for $\oplus$ and $\times$ for $\otimes$. Pointwise function h encodes the color state update logic in the Apply function: $h(x, y) = (y == 0) \times x + (1 - (y == 0)) \times ((y+1)\,\&\, \sim y)$, which returns x if y is 0 and $(y+1)\,\&\, \sim y$ otherwise. Here, & is bitwise AND operation, $\sim$ is bitwise NOT operation.

We next justify why AutoMI populates GC with TrackFree. First, e_{GC} is active idempotent. Indeed, observe that for any $\lambda_{\text{msg\_acc}}$ as the bitvec representing the color state aggregated from neighbors, (a) if $\lambda_{\text{msg\_acc}} = 0$, we have $e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle = \mathbf{x}_{\text{ans}}$ and $e_{\text{ans}}[e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle]\langle\lambda_{\text{msg\_acc}}\rangle = e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle = \mathbf{x}_{\text{ans}}$; (b) if $\lambda_{\text{msg\_acc}} \neq 0$, let c be the available minimum color, $e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle = \mathbf{c}$ and $e_{\text{ans}}[e_{\text{ans}}[\mathbf{x}_{\text{ans}}]\langle\lambda_{\text{msg\_acc}}\rangle]\langle\lambda_{\text{msg\_acc}}\rangle = \mathbf{c}$. To see e_{GC} is inactive idempotent, we observe that initially non-source vertex has no color assigned, *i.e.,* the initialized msg_acc value $\alpha_{\text{msg\_acc}} = 0$. As a result, $e_{\text{GC}}[\mathbf{x}_{\text{ans}}]\langle\alpha_{\text{msg\_acc}}\rangle = \mathbf{x}_{\text{ans}}$. That is, e_{GC} is also inactive idempotent. Since $\delta_{\text{GC}} :=$ **changed** is regular, by Theorem 2 we know that GC [24] is TrackFree.

Table 4. Real-life data graphs

Graph	#vertices $\|V\|$	#edges $\|E\|$
`LiveJournal` [5]	4,847,571	68,993,773
`Twitter` [11]	41,652,230	1,468,365,182
`Friendster` [1]	68,349,466	2,586,147,869
`UKDomain` [12]	105,153,952	3,301,876,564
`MovieLens` [6]	165,771	20,000,263
`Netflix` [8]	497,959	100,480,507

Advanced graph analytics. AutoMI also generates multi-instance version of advanced graph analytics, including Sparse Matrix-Vector multiplication (SpMV), Personalized PageRank (PPR), and Collaborative Filtering (CF). We follow [21, 32, 34, 49, 53] to reduce SpMV to a graph traversal problem; similarly for PPR as an extension of SpMV. CF uses Stochastic Gradient Descent [31] to iteratively update latent factors of vertices being visited, starting from a source.

Both SpMV and PPR are `TrackFree` by Theorem 2 and hence AutoMI generates their multi-instance version with `TrackFree`, while CF is not `TrackFree`. We omit the details; however, we implement and evaluate their performance in Section 7.

7 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we evaluated the performance of multi-instance GAS algorithms produced by AutoMI. We first specify the setup (Section 7.1), then give the evaluation results (Sections 7.2-7.4) and finally summarize our findings (Section 7.5).

7.1 System Implementation and Experimental Setup

System. AutoMI uses libclang [4] to parse single-instance GAS program C++ source code into an abstract syntax tree (AST) and traverse the AST with cursors. It performs auto-conversion in two steps as described in Section 3.2, *i.e.,* AST annotation and vectorized code generation. AutoMI produces vectorized multi-instance GAS algorithm in C++, with `TrackFree` enabled when it was applicable according to the analysis Sections 4-5. It also implements serialization methods for the populated data structures (vertex properties), including track, which are needed by GAS systems for message passing. GAS programs generated by AutoMI run on popular GAS systems, *e.g.,* PowerGraph [24] and PowerLyra [20]. By default, AutoMI uses PowerLyra to run the converted GAS programs as it is more current; however, we remark that AutoMI is generic and portable, and can be deployed on any standard GAS systems.

To support SIMD vectorization, AutoMI uses aligned memory layout. It populates each vertex property by collocating values for k sources into a k-size array. Using this data layout enhances spatial locality in its multi-instance GAS programs. To utilize Intel SIMD intrinsics [3] for vectorized code generation, AutoMI maintains a mapping from logical and arithmetical operators to their corresponding SIMD intrinsics. If a data type or associated operator is not found in the mapping, AutoMI generates "abstract" SIMD operators to support vectorization. For instance, it vectorizes bitvec (*e.g.,* Algorithm 5) into array of bitvec and overloads the classic SIMD operators accordingly, such that SIMD intrinsics are applied iteratively over the array, processing 256 bits each time.

Compared systems. We compared the performance of AutoMI populated multi-instance GAS algorithms with the following methods.

(a) PowerLyra [20]. A popular GAS system for distributed graph processing, which improves upon PowerGraph [24] via its hybrid graph partitioning method. We used its open-source implementation

Table 5. Average speedup of all methods over PowerLyra; ✗ marks algorithms not available

system	bBFS	iBFS	SSSP	GC	SpMV	PPR	CF
PowerLyra [20]	1	1	1	1	1	1	1
Quegel [59, 64]	2.32x	2.25x	1.77x	✗	✗	✗	✗
MultiLyra [40]	6.07x	5.53x	5.16x	✗	✗	✗	✗
AutoMI	**14.82x**	**9.51x**	**14.09x**	**6.37x**	**7.79x**	**8.21x**	**3.78x**

from [9]. To compute answers for multiple source vertices, PowerLyra invokes a single-instance GAS algorithm for each of the sources one by one, each time using all the working threads.

(b) MultiLyra [40]. A generalization of PowerLyra system that supports batched evaluation of multiple query instances. We used its built-in multi-instance algorithms implemented by the authors.

(c) Quegel [59, 64]. A distributed graph system for batched query processing, with Pregel's vertex-centric programming interface. We used its implementation from [10]. While Quegel is designed for point-to-point queries, we adapted its vertex-centric programs to evaluate point-to-all queries.

Since AutoMI, PowerLyra and MultiLyra employ the same GAS engine (*i.e.,* PowerLyra) to execute vertex-centric computation, the evaluation could accurately reflect and fairly compare the quality of (a) serial evaluation of multi-instance algorithms by PowerLyra, (b) handcrafted optimized implementation of multi-instance algorithms with MultiLyra, and (c) the multi-instance GAS program generated by AutoMI, excluding the noise of vertex-centric engines. We also compared AutoMI with Quegel's batching technique for processing multiple query instances, to evaluate the effectiveness of vectorization for multi-instance computation in general.

Algorithms. We tested the performance of Integer BFS (iBFS), Boolean BFS (bBFS) and shortest path (SSSP) using all four systems, and evaluated Greedy Graph Coloring (GC), SpMV, PPR and Collaborative Filtering (CF) with AutoMI and PowerLyra only, as they are not available in MultiLyra and Quegel. For MultiLyra, we use the implementation of multi-instance BFS and SSSP from its authors. AutoMI uses the multi-instance GAS programs it generated as described in Sections 3-6, which are then evaluated on PowerLyra.

Graphs. We used 6 real-life graphs of varying sizes (Table 4), among which MovieLens and Netflix are bipartite graphs for collaborative filtering (CF). For the other 4 graphs, following [37, 52], we generated edge weights between $[1, \log |V|)$ uniformly at random.

Queries. Following [54, 61], we randomly sampled a seed vertex and run a BFS starting from the seed vertex to get k source 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. In each test, we used the same source vertices as input for all compared methods. To reduce noise, we sampled three such seed vertices and generated three groups of queries for each data graph; the average is reported.

Configuration. The experiments were run on a cluster of four r6i.8xlarge AWS EC2 instances, interconnected by a 12.5 Gbps network. Each EC2 instance has 32 vCPUs and 256 GB of memory. Each experiment was run 3 times; the average is reported. For a fair comparison, we used the same GAS system configurations and hybrid graph partitioning for AutoMI, PowerLyra and MultiLyra.

7.2 Overall Performance

Varying the number k of sources from 16 to 256, we tested the run time of all methods over all the graphs. The results are depicted in Figures 3-4 and Table 5, and tell us the following.

(1) AutoMI is effective, and is consistently the best among all for each and every case tested. As

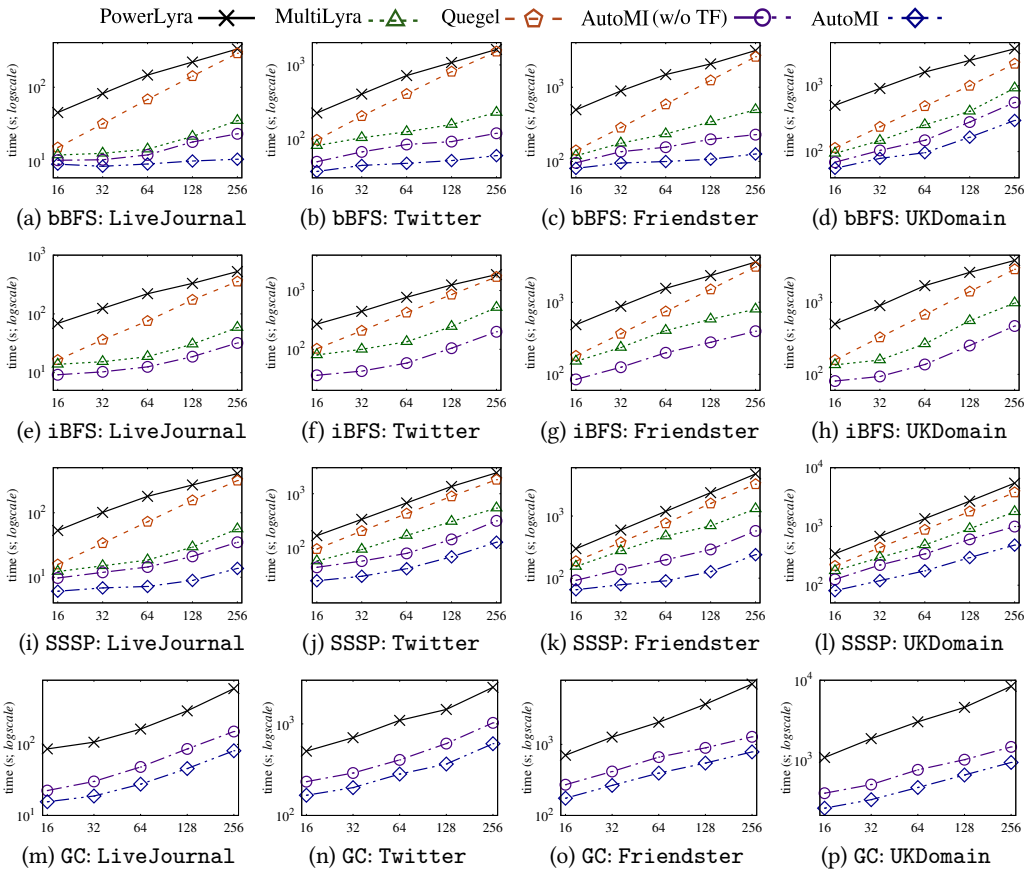


Fig. 3. Running time of all methods with varying number (k) of queries (Section 7.2)

shown in Table 5, multi-instance GAS programs produced by AutoMI significantly outperform the serial evaluation by PowerLyra, highly optimized MultiLyra code and Quegel batch evaluation for all cases. It is up to 27.83x, 16.42x, 29.51x, 9.84x, 10.26x, 10.65x and 6.17x faster than PowerLyra for bBFS, iBFS, SSSP, GC, SpMV, PPR and CF, respectively. Compared to MultiLyra and Quegel, AutoMI is up to 3.82x and 26.35x faster for bBFS, 2.57x and 7.12x for iBFS, 4.63x and 23.01x for SSSP.

The improvement over MultiLyra is due to the use of vectorization in AutoMI converted programs, which MultiLyra does not capitalize on. This further justifies the value of vectorization and its associated challenges that AutoMI has relieved from developers. We observe that Quegel is outpefromed by AutoMI and MultiLyra. This is because Quegel does not integrate messages across query instances, missing out on the opportunity to amortize communication cost. In addition, Quegel does not use SIMD vectorization, which is shown to significantly improve performance in AutoMI.

(2) The speedup of AutoMI over PowerLyra, MultiLyra and Quegel increases when more queries (sources) are used. As shown in Fig. 3c, AutoMI Boolean BFS is 6.17x, 1.85x and 2.71x faster than PowerLyra, MultiLyra and Quegel with 16 queries over Friendster, respectively, up to 23.74x, 3.56x and 19.55x with 256 queries; similarly for other algorithms and graphs. Quegel's performance drops when more queries are used. The per-query amortized runtime of Quegel bBFS is 8.79s with 16 queries over Friendster, while this increases to 10.06s with 256 queries. This is in consistent with

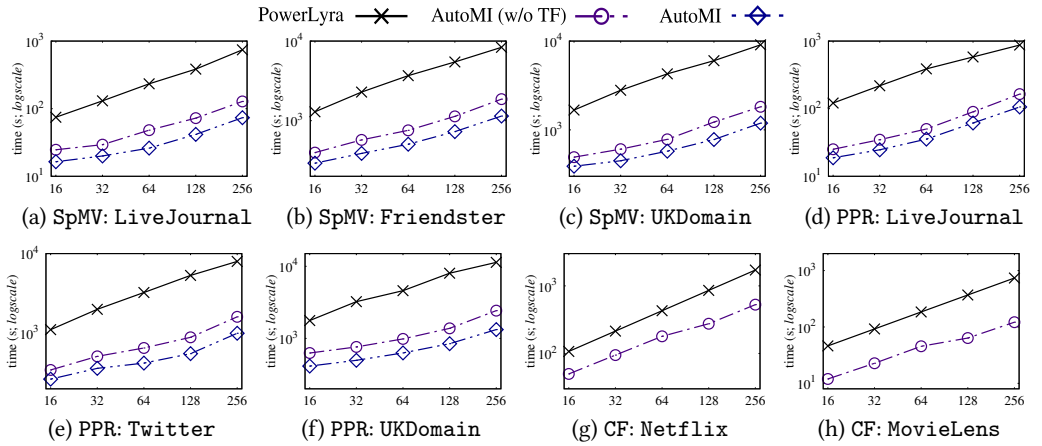


Fig. 4. Running time of SpMV, PPR and CF with varying number (k) of queries (Section 7.2)

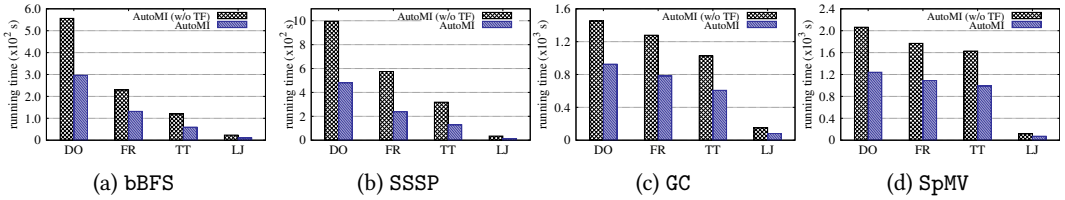


Fig. 5. Effectiveness of TrackFree optimization in AutoMI (Section 7.3)

[59], which states that Quegel's batching technique is designed for point-to-point queries that access a small portion of graphs, and is not optimized for larger number of concurrent query instances.

(3) AutoMI is more effective for TrackFree algorithms. For instance, over LiveJournal, for the TrackFree SSSP, AutoMI is on average 21.65x and 2.83x faster than PowerLyra and MultiLyra, respectively, compared to 14.23x and 1.59x for iBFS that is not TrackFree.

7.3 Breakdown

To understand how AutoMI gains its speedups, we further evaluated the effectiveness of AutoMI vectorization and TrackFree optimization, by examining its run time breakdown and the total number of messages (#comms) collected at runtime. The results for SSSP over UKDomain with 256 queries are shown in Table 6; here AutoMI (w/o TF) denotes AutoMI with TrackFree disabled.

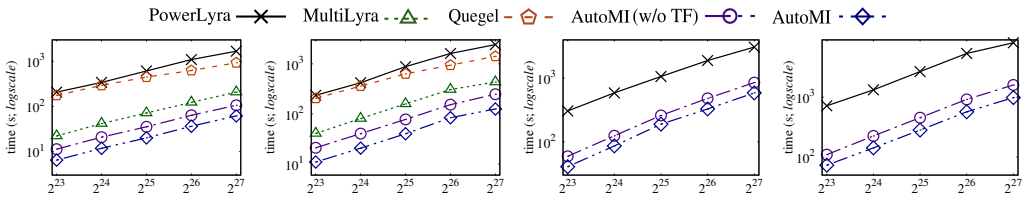
Observe that multi-instance methods (MultiLyra, AutoMI (w/o TF) and AutoMI) significantly reduce #comms and speed up the run time of each phase over PowerLyra; AutoMI is the fastest of all.

SIMD Vectorization. Vectorization provides evident performance boost. Overall AutoMI (w/o TF) is 1.78x faster over MultiLyra. Indeed, AutoMI (w/o TF) is 1.82x, 1.87x and 1.31x faster than MultiLyra in Gather, Apply and Scatter phase, respectively. This confirms the benefit of vectorization.

TrackFree. TrackFree further improves AutoMI on top of vectorization. When TrackFree is enabled, AutoMI is 2.07x faster than AutoMI (w/o TF), with 2.28x, 2.13x and 1.67x speedup in Gather, Apply and Scatter phase, respectively. This is because TrackFree optimization eliminates the overhead of track reference and maintenance throughout Gather, Apply and Scatter phases. More specifically, TrackFree optimization speeds up AutoMI over AutoMI (w/o TF) by 2.21x, 2.24x, 1.69x, 1.76x and 1.83x for Boolean BFS (bBFS), SSSP, GC, SpMV and PPR, respectively, as shown in Fig. 5.

Table 6. Speedup over PowerLyra and breakdown for SSSP over UKDomain with 256 queries

system	speedup	Gather (s)	Apply (s)	Scatter (s)	Sync (s)	#comms
PowerLyra	1	1787.14	1324.96	2033.49	304.67	4.61×10^{10}
MultiLyra	3.07x	801.48	777.87	183.55	12.25	4.73×10^{8}
AutoMI (w/o TF)	5.46x	438.89	416.98	139.79	12.24	4.73×10^{8}
AutoMI	11.32x	192.84	196.16	83.68	11.81	4.73×10^{8}



(a) bBFS: rMat (b) SSSP: rMat (c) GC: rMat (d) SpMV: rMat

Fig. 6. Scalability over synthetic graphs with varying number $|V|$ of vertices (Section 7.4)

SIMD Vectorization + TrackFree. Putting them together, when running SSSP on UKDomain with 256 queries, AutoMI is 11.32x and 3.69x faster than PowerLyra and MultiLyra, respectively.

7.4 Scalability

Using rMat [18], we generated synthetic graphs of varying sizes to evaluate the scalability of AutoMI. The results with 256 sources are shown in Fig. 6, and tell us the following. (1) AutoMI consistently outperforms PowerLyra, MultiLyra and Quegel in all cases. (2) The delicate computation and optimization that AutoMI embeds into multi-instance programs do not impair or hinder their scalability. Indeed, the speedup ratio over the compared systems is stable as graph grows, *e.g.*, for Boolean BFS, AutoMI is 27.31x, 3.36x and 17.65x faster than PowerLyra, MultiLyra and Quegel, respectively, over rMat with 2^{23} vertices, and it remains 27.52x, 3.38x and 17.21x faster with 2^{27} vertices.

7.5 Summary

We find that interleaved evaluation of multiple query instances on distributed GAS system is efficient. On average AutoMI is 14.82x, 9.51x, 14.09x, 6.37x, 7.79x, 8.21x and 3.78x faster than PowerLyra for Boolean BFS, Integer BFS, SSSP, GC, SpMV, PPR and CF, respectively. In addition, AutoMI receives effective performance improvement from its automated SIMD vectorization and TrackFree optimization, and AutoMI is scalable as graph size increases.

8 CONCLUSION

We have proposed AutoMI, an approach to automatically converting single-instance GAS algorithms into optimized multi-instance version that can run on any generic vertex-centric GAS system. We have discussed its design, proved its correctness, and developed track-free vectorization, an effective optimization with an algebraic characterization that allows AutoMI to make full and safe use of it.

We are currently extending AutoMI with support for (a) vertex-centric systems with a non-GAS programming interface and (b) concurrent iterative graph frameworks to allow concurrent multi-instance computation via thread grouping and scheduling.

ACKNOWLEDGMENTS

This work is supported by RAEng RF\201920\19\319 and the Huawei-Edinburgh Joint Lab.

REFERENCES

[1] Access: 2024. Friendster. http://konect.cc/networks/friendster/.
[2] Access: 2024. Giraph. https://giraph.apache.org/.
[3] Access: 2024. Intel® Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.
[4] Access: 2024. libclang. https://clang.llvm.org/.
[5] Access: 2024. LiveJournal. http://snap.stanford.edu/data/soc-LiveJournal1.html.
[6] Access: 2024. MovieLens. http://grouplens.org/datasets/movielens/.
[7] Access: 2024. Multi-instance BFS implementation. https://github.com/mtodat/ms-bfs.
[8] Access: 2024. Netflix. http://konect.cc/networks/netflix/.
[9] Access: 2024. PowerLyra. https://github.com/realstolz/powerlyra/.
[10] Access: 2024. Quegel. http://www.cse.cuhk.edu.hk/systems/quegel/.
[11] Access: 2024. Twitter. http://konect.cc/networks/twitter/.
[12] Access: 2024. UK domain. http://konect.cc/networks/dimacs10-uk-2007-05/.
[13] Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2024. Convergence of datalog over (pre-) semirings. *J. ACM* 71, 2 (2024), 1–55.
[14] Zahid Abul-Basher. 2017. 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*. IEEE Computer Society, 1426–1430.
[15] Khaled Ammar and M Tamer Özsu. 2018. Experimental analysis of distributed graph systems. *Proc. VLDB Endow.* 11, 10 (2018), 1151–1164.
[16] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. 2003. Navigation- vs. Index-Based XML Multi-Query Processing. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman (Eds.). IEEE Computer Society, 139–150.
[17] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*. Springer, 316–330.
[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.
[19] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. 2021. Krill: a compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.
[20] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.
[21] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Exploiting in-Hub Temporal Locality in SpMV-based Graph Processing. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, 42:1–42:10.
[22] Haishuang Fan, Ming Li, Jingya Wu, Wenyan Lu, Xiaowei Li, and Guihai Yan. 2023. BitColor: Accelerating Large-Scale Graph Coloring on FPGA with Parallel Bit-Wise Engines. In *Proceedings of the 52nd International Conference on Parallel Processing*. 492–502.
[23] Floris Geerts, Thomas Muñoz, Cristian Riveros, and Domagoj Vrgoč. 2021. Expressive Power of Linear Algebra Query Languages. In *PODS*.
[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.
[25] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.
[26] Todd J Green and Val Tannen. 2017. The semiring framework for database provenance. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 93–99.
[27] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.* 7, 12 (2014), 1047–1058.
[28] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 349–362.
[29] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2011. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.* 27, 2 (2011), 303–325.
[30] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. 2017. Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs. In *Proceedings of the 20th International Conference on*

Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 1–12.

[31] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[32] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46.

[33] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable Multi-query Optimization for SPARQL. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 666–677.

[34] Jia Li, Wenyue Zhao, Nikos Ntarmos, Yang Cao, and Peter Buneman. 2023. MITra: A Framework for Multi-Instance Graph Traversal. *Proceedings of the VLDB Endowment* 16, 10 (2023).

[35] Xinyi Liu, Zhigang Wang, Ning Wang, Xiangtan Li, Bo Zhang, Jun Qiao, Zhiqiang Wei, and Jie Nie. 2021. An Adaptive Sharing Framework for Efficient Multi-source Shortest Path Computation. In *Web Information Systems and Applications - 18th International Conference, WISA 2021, Kaifeng, China, September 24-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12999)*, Chunxiao Xing, Xiaoming Fu, Yong Zhang, Guigang Zhang, and Chaolemen Borjigin (Eds.). Springer, 635–646.

[36] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012).

[37] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. 2021. Cache-Efficient Fork-Processing Patterns on Large Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1208–1221.

[38] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: an experimental evaluation. *Proc. VLDB Endow.* 8, 3 (2014), 281–292.

[39] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 135–146.

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

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

[42] Mehryar Mohri et al. 2002. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics* 7, 3 (2002), 321–350.

[43] Peitian Pan and Chao Li. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*. IEEE Computer Society, 217–224.

[44] Peng Peng, Qi Ge, Lei Zou, M. Tamer Özsu, Zhiwei Xu, and Dongyan Zhao. 2021. Optimizing Multi-Query Evaluation in Federated RDF Systems. *IEEE Trans. Knowl. Data Eng.* 33, 4 (2021), 1692–1707.

[45] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*.

[46] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2012. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 375–394.

[47] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2015. Synthesizing parallel graph programs via automated planning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 533–544.

[48] Xuguang Ren and Junhu Wang. 2016. Multi-Query Optimization for Subgraph Isomorphism Search. *Proc. VLDB Endow.* 10, 3 (2016), 121–132.

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

[50] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (1988), 23–52.

[51] Timos K. Sellis and Subrata Ghosh. 1990. On the Multiple-Query Optimization Problem. *IEEE Trans. Knowl. Data Eng.* 2, 2 (1990), 262–266.

[52] Julian Shun and Guy E. Blelloch. [n. d.]. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.).

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

[54] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.* 8, 4 (2014), 449–460.

[55] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proceedings of the VLDB Endowment* 10, 8 (2017), 877–888.

[56] Xinmin Tian, Hideki Saito, Serguei Preis, Eric N. Garcia, Sergey Kozhukhov, Matt Masten, Aleksei G. Cherkasov, and Nikolay Panchenko. 2013. Practical SIMD Vectorization Techniques for Intel® Xeon Phi Coprocessors. In *IPDPS Workshops*.

[57] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: an efficient, low-cost system for concurrent graph processing. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 227–238.

[58] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. 2017. Big graph analytics platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195.

[59] Da Yan, James Cheng, M. Tamer Özsu, Fan Yang, Yi Lu, John C. S. Lui, Qizhen Zhang, and Wilfred Ng. 2016. A General-Purpose Query-Centric Framework for Querying Big Graphs. *Proc. VLDB Endow.* 9, 7 (2016), 564–575.

[60] Da Yan, James Cheng, M. Tamer Özsu, Fan Yang, Yi Lu, John C. S. Lui, Qizhen Zhang, and Wilfred Ng. 2016. A General-Purpose Query-Centric Framework for Querying Big Graphs. *Proc. VLDB Endow.* 9, 7 (2016), 564–575.

[61] Hiroki Yanagisawa. 2010. 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*. 1–10.

[62] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2022. Glign: Taming misaligned graph traversals in concurrent graph processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 78–92.

[63] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. 2017. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*. 40–51.

[64] Qizhen Zhang, Da Yan, and James Cheng. 2016. Quegel: A General-Purpose System for Querying Big Graphs. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, 2189–2192.

[65] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 441–452.

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

[67] Jin Zhao, Yu Zhang, Ligang He, Qikun Li, Xiang Zhang, Xinyu Jiang, Hui Yu, Xiaofei Liao, Hai Jin, Lin Gu, et al. 2023. GraphTune: An Efficient Dependency-aware Substrate to Alleviate Irregularity in Concurrent Graph Processing. *ACM Transactions on Architecture and Code Optimization* (2023).

[68] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, 3:1–3:14.

Received April 2024; revised July 2024; accepted August 2024