RingSG: Optimal Secure Vertex-Centric Computation for Collaborative Graph Processing

ZHENHUA ZOU, Tsinghua University, Beijing, China

ZHUOTAO LIU, Tsinghua University, Beijing, China

JINYONG SHAN, Beijing Smartchip Microelectronics Technology Co., Ltd., Beijing, China

QI LI, Tsinghua University, Beijing, China

KE XU, Tsinghua University, Beijing, China

MINGWEI XU, Tsinghua University, Beijing, China

Published: 19 November 2025

Citation in BibTeX format

CCS '25: ACM SIGSAC Conference on Computer and Communications Security
October 13 - 17, 2025
Taipei, Taiwan

Conference Sponsors:
SIGSAC

RingSG: Optimal Secure Vertex-Centric Computation for Collaborative Graph Processing

Zhenhua Zou
Tsinghua University
Beijing, China
zou-zh21@mails.tsinghua.edu.cn

Zhuotao Liu*
Tsinghua University
& State Key Laboratory of Internet Architecture
Beijing, China
zhuotaoliu@tsinghua.edu.cn

Jinyong Shan
Beijing Smartchip Microelectronics Technology Co., Ltd.
Beijing, China
shanjinyong@sgchip.sgcc.com.cn

Qi Li
Tsinghua University
& State Key Laboratory of Internet Architecture
Beijing, China
qli01@tsinghua.edu.cn

Ke Xu
Tsinghua University
& State Key Laboratory of Internet Architecture
Beijing, China
xuke@tsinghua.edu.cn

Mingwei Xu
Tsinghua University
& State Key Laboratory of Internet Architecture
Beijing, China
xumw@tsinghua.edu.cn

Abstract

Collaborative graph processing refers to the joint analysis of interconnected graphs held by multiple graph owners. To honor data privacy and support various graph processing algorithms, existing approaches employ secure multi-party computation (MPC) protocols to express the vertex-centric abstraction. Yet, due to certain computation-intensive cryptography constructions, state-of-the-art (SOTA) approaches are asymptotically suboptimal, imposing significant overheads in terms of computation and communication. In this paper, we present RingSG, the first system to attain optimal communication/computation complexity within the MPC-based vertex-centric abstraction for collaborative graph processing. This optimal complexity is attributed to Ring-ScatterGather, a novel computation paradigm that can avoid exceedingly expensive cryptography operations (e.g., oblivious sort), and simultaneously ensure the overall workload can be optimally decomposed into parallelizable and mutually exclusive MPC tasks. Within Ring-ScatterGather, RingSG improves the concrete runtime efficiency by incorporating 3-party secure computation via share conversion, and optimizing the most cost-heavy part using a novel oblivious group aggregation protocol. Finally, unlike prior approaches, we instantiate RingSG into two end-to-end applications to effectively obtain application-specific results from the protocol outputs in a privacy-preserving manner. We developed a prototype of RingSG and extensively evaluated it across various graph collaboration settings, including different graph sizes, numbers of parties, and average vertex degrees. The results show RingSG reduces the system running time of SOTA approaches by up to 15.34× and per-party communication by up to 10.36×. Notably, RingSG excels in processing sparse global graphs collectively held by more parties, consistent with our theoretical cost analysis.

CCS Concepts

• **Security and privacy → Cryptography**; • **Computing methodologies → Distributed algorithms**.

Keywords

Collaborative Graph Processing; Secure Multi-party Computation; Vertex-Centric Computation

ACM Reference Format:
Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. 2025. RingSG: Optimal Secure Vertex-Centric Computation for Collaborative Graph Processing. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744824

1 Introduction

Graph data, which constitutes a fundamental element of modern data infrastructure, often exhibits complex interdependencies across organizations. A prime example is the banking sector, where inter-bank transfers function as *inter-edges* connecting individual banks' transfer graphs into a comprehensive *global graph*. This interconnected structure necessitates *collaborative graph processing* [5, 6, 10, 18, 34, 38], which enables multiple organizations to jointly analyze their interconnected graphs and derive insights that would be unattainable through isolated analysis of individual graphs. Anti-money laundering (AML) [16, 17, 39] represents a particularly significant application of such joint analysis. The detection of malicious cross-border fund flows [44] becomes infeasible when relying solely on isolated local graph data maintained by individual banks. However, the growing emphasis on data privacy presents a significant challenge, as direct sharing of graph data among different graph owners for collaborative processing raises substantial privacy concerns and may violate regulatory requirements [1].

*Zhuotao Liu is the corresponding author.

Secure Multi-party Computation (MPC) [9, 13, 15, 21, 29, 33, 40], a suite of cryptographic protocols, offers a promising solution by enabling collaborative graph processing with formally provable privacy guarantees. Most existing MPC-based approaches employ the fully outsourced computation setting, where the graph owners employ several third-party computing servers to hold and compute their private graph data in a secret-shared fashion. These approaches either treat MPC as an arithmetic black box (ABB) and focus on optimizing specific graph algorithms (like Shortest Path [5–7, 22, 35]), or depend on specific MPC protocols (like Garbled circuit, ABY3) to support general computation of various graph algorithms [10, 12, 23, 27, 28, 34]. The latter branch is largely based on GraphSC [34], a secure computation paradigm to execute various graph algorithms under the *vertex-centric abstraction* [26].

However, the outsourced computation scheme exhibits two fundamental limitations. (i) It necessitates a one-time, yet exceedingly costly, secure sort to rearrange the graph data, which can consume up to 75% or more of the total execution time [10]. (ii) Since each computing party retains an essential secret share of the global graph, it is obligated for each of them to perform secure computation across the entire graph, even if certain sections of the global graph are topologically disconnected from the party. This results in substantial communication overhead between each pair of parties.

To address the above problem, CoGNN [41] proposes to integrate distributed computing into the vertex-centric abstraction. Its core idea is to decompose the overall secure computation workload into concurrent two-party secure computation (2PC) tasks. This allows each party to process only part of the global graph and to replace the costly secure sort with more computation-efficient secure permutations [14, 31]. However, CoGNN only achieves *sub-optimal asymptotic complexity* in the vertex-centric abstraction because it fails to attain optimal computation task partitioning. Consequently, the proportion of global graph data associated with different tasks overlaps, leading to redundant computations. Moreover, the secure execution of the decomposed tasks is confined to 2PC, which lacks the generality necessary to utilize more efficient MPC protocols. See detailed discussion of CoGNN in § 3.2.

To advance the state-of-the-art (SOTA), we introduce RingSG, the first collaborative graph processing system that attains an *optimal complexity* within the vertex-centric abstraction. RingSG demonstrates several key advantages over the SOTA approaches. First, it introduces Ring-ScatterGather, a novel computation paradigm that introduces *grouped data parallelism* into the vertex-centric abstraction. Unlike the outsourced computation schemes and CoGNN, Ring-ScatterGather organizes the overall secure computation workload into *rings of parallelizable and non-overlapping tasks*, where each task is securely computed by a group of relevant parties, as shown in Figure 1. As a result, Ring-ScatterGather achieves the communication complexity of $O(|V| + |E|)$ in the vertex-centric abstraction, where $|V|$ and $|E|$ represent the numbers of vertices and edges in the global graph, respectively. This is *optimal* because it is linear to $(|V|+|E|)$ and independent of the number of parties N, making it more efficient than both outsourced computation schemes ($O((|V|+|E|)\log(|V|+|E|))$) and CoGNN ($O(N|V|+|E|)$). We summarize the communication/round complexities of RingSG and prior SOTAs in Table 1. For GraphSC, we use its SOTA construction [23].

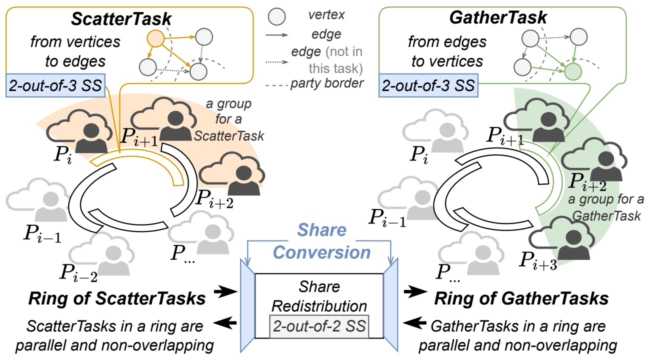


Figure 1: A high-level illustration of RingSG. (i) MPC-based vertex-centric computation is decomposed into parallel Scatter/Gather tasks, each assigned to a group of parties. (ii) Tasks in an iteration of vertex-centric computation are non-overlapping and parallelized like a ring. (iii) Synchronization occurs between rings, and share conversion/redistribution is performed during the transition between consecutive rings.

	Communication (Bandwidth)	Round
GraphSC [23]	$O((\|V\| + \|E\|)\log(\|V\| + \|E\|))$	$O(\log(\|V\| + \|E\|))$
CoGNN [41]	$O(N\|V\| + \|E\|)$	$O(\log(\|E\|) + N)$
RingSG	$\boldsymbol{O(\|V\| + \|E\|)}$	$\boldsymbol{O(\log(\|E\|))}$

Table 1: The overall communication and round complexity for executing one iteration of vertex-centric abstraction.

Second, unlike CoGNN relying on relatively expensive 2PC to execute these decomposed MPC tasks, RingSG integrates efficient three-party secure computation (3PC). This is accomplished through a *share conversion* and a *share redistribution* mechanism, within the Ring-ScatterGather paradigm (also shown in Figure 1). This mechanism dynamically establishes three-party replicated secret shares on demand, enhancing task execution efficiency while ensuring security. Meanwhile, we pinpoint the most computationally intensive component in RingSG: obliviously aggregating the edge-produced data targeting the identical vertices. To reduce its overhead, we propose a novel cryptographic protocol to attain Oblivious Group Aggregation (OGA) with identical computation/communication as the SOTA [8, 19, 20], but with halved rounds. This further reduces the concrete running time of RingSG.

Finally, we present the end-to-end instantiation of RingSG in two real-world anti-money laundering applications. We demonstrate that the design of RingSG has enabled efficient and privacy-preserving extraction of application-specific results from the protocol outputs, which has not been discussed in existing approaches.

Contributions. Our primary contribution is RingSG, the first system that realizes the optimal communication/computation complexity for the MPC-based vertex-centric abstraction in collaborative graph processing. The optimal complexity is enabled by a novel computation paradigm (Ring-ScatterGather) that decomposes the holistic secure computation workload into parallelizable and non-overlapping tasks, resulting in superior efficiency and scalability. Within this paradigm, we incorporate 3PC based on replicated secret share, and also propose a new OGA protocol to further improve the concrete efficiency of RingSG. Finally, we discuss the

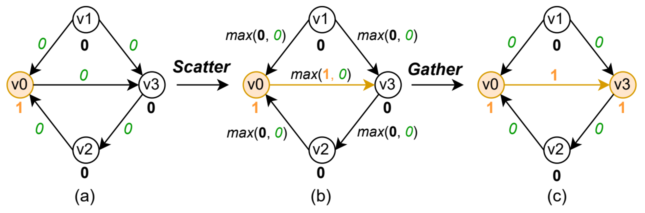


Figure 2: An example showing an iteration of vertex-centric abstraction for Connected Component Labeling (CC).

end-to-end instantiations of RingSG to confidentially and efficiently obtain application-specific results from the protocol outputs. We implement RingSG and experimentally compare it to the SOTA approaches across a wide range of settings, including different graph sizes, numbers of parties and average vertex degrees. Evaluation results show that RingSG reduces the overall system running time of SOTA approaches by up to 15.34× and per-party communication by up to 10.36×. For sparse global graphs with a fixed average vertex degree, RingSG exhibits a performance gain that grows as the number of parties or graph size increases. This observation aligns with our theoretical analysis, highlighting RingSG's superior asymptotic complexity and concrete efficiency advantages.

2 Preliminaries

Graph. A graph can be specified as $G = (V, E)$, where V is a vector of vertices and E is a vector of edges. Each slot of V contains a vertex $v = (v.\text{id}, v.\text{data})$, where v.id is a unique identifier and v.data is the vertex data. Each slot of E stores a directed edge $e = (e.\text{src}, e.\text{dst}, e.\text{data})$, where e.src (e.dst) is the identifier of the source (destination) vertex. We call edge e the outgoing edge of e.src and the incoming edge of e.dst. The number of outgoing (incoming) edges of a vertex is called its outgoing (incoming) degree. We use V.id to represent the identifier vector of V, and use E.src (E.dst) for the identifier vector of E's source (destination) vertices.

Graph Processing via Vertex-Centric Abstraction. Graph processing essentially leverages graph algorithms to analyze a graph, and produces either updated graph data or statistics extracted from it. As proposed in Pregel [24, 26], most graph algorithms can be efficiently and parallelly computed via three vertex-centric operations: Gather, Apply and Scatter. This is called the *vertex-centric abstraction*. For classical graph algorithms like Connected Component Labeling (CC) [2], Shortest Path (SP) [4] and PageRank (PR) [3], Apply can be included in Gather and the graph algorithms are then expressed as *iterations* of Scatter-Gather, as follows:

- *Scatter.* For all $v \in V$, traverse its outgoing edges, combining the vertex data and edge data to generate an update on the edge: $u \leftarrow \mathcal{F}_S(v, e), \forall e \in \{e|e.\text{src} = v.\text{id}\}$.
- *Gather.* For all $v \in V$, aggregate the updates produced on each of its incoming edges to produce the updated vertex data: $v' \leftarrow \mathcal{F}_G(v, u), \forall u \in \{u|u.\text{dst} = v.\text{id}\}$.

$\mathcal{F}_S$ and $\mathcal{F}_G$ are graph algorithm-specific. Figure 2 illustrates a single iteration of Scatter-Gather for CC, detecting downstream vertices and edges connected to vertex v0. Bold numbers under each vertex (v.data) and italic numbers over each edge (e.data) represent their current labels. A label of "1" indicates a connection.

Functionality $\langle T' \rangle \leftarrow \mathcal{F}_{\text{OEP}}$ $(\langle T \rangle, \pi)$

Input: 1. $\mathcal{F}_{\text{OEP}}$ receives $\langle T \rangle_0$ from P_i and $\langle T \rangle_1$ from P_j;

2. $\mathcal{F}_{\text{OEP}}$ receives π from P_i, where $\pi : \mathbb{Z}_{|T'|} \to \mathbb{Z}_{|T|}$. π can be specified by a pair of vectors (src, dst), $|\text{src}| = |T|, |\text{dst}| = |T'|$.

Output: 1. $\mathcal{F}_{\text{OEP}}$ sends $\langle T' \rangle_0$ to P_i, where $T'[x] := T[\pi(x)]$;

2. $\mathcal{F}_{\text{OEP}}$ sends $\langle T' \rangle_1$ to P_j, where $T'[x] := T[\pi(x)]$.

Functionality 1: $\mathcal{F}_{\text{OEP}}$ for Oblivious Extended Permutation

Functionality $\langle T' \rangle \leftarrow \mathcal{F}_{\text{OGA}}$ $(\langle T \rangle, \mathcal{G}, \mathcal{F}_{\boxplus})$

Input: 1. $\mathcal{F}_{\text{OGA}}$ receives $\langle T \rangle_0$ from P_i and $\langle T \rangle_1$ from P_j;

2. $\mathcal{F}_{\text{OGA}}$ receives $\mathcal{G}$ from P_i, where $|\mathcal{G}| = |T|$ and $\mathcal{G}$ can be divided into segments $g_0||...||g_{m-1}$, each containing elements of the same value. $\mathcal{F}_{\text{OGA}}$ receives a binary merging operation $\mathcal{F}_{\boxplus}$.

Output: $\mathcal{F}_{\text{OGA}}$ sends $\langle T' \rangle_0$ and $\langle T' \rangle_1$ to P_i and P_j respectively, such that if x_k is the start index of $g_k, k \in [m]$, then $\langle T' \rangle[x_k]$ stores the $\mathcal{F}_{\boxplus}$-aggregated result of g_k's corresponding elements in $\langle T \rangle$.

Functionality 2: $\mathcal{F}_{\text{OGA}}$ for Oblivious Group Aggregation

- *Scatter*: The maximum value of v.data and e.data (from outgoing edges) is assigned to u on the edge (Figure 2(b)).
- *Gather*: The u on each edge updates the destination vertex's label, assigning the maximum of u and v.data to v.data$'$.

After this iteration, v3's label updates while v1 and v2 remain unchanged (Figure 2(c)), indicating v3 is connected to v0.

Fixed-Point Encoding and Secret Share. In RingSG, we encode all the graph data as a Fixed-Point representation over the ring, $\mathbb{Z}_L$, where $L := 2^l$. RingSG leverages two secret share schemes over $\mathbb{Z}_L$, i.e., 2-out-of-2 and 2-out-of-3 additive secret share. In particular, the 2-out-of-2 secret share of $x \in \mathbb{Z}_L$ is denoted as $x \equiv \langle x \rangle_0 + \langle x \rangle_1 \pmod{L}$, where $\langle x \rangle_0$ is sampled uniformly and $\langle x \rangle_0, \langle x \rangle_1$ are held by two different parties, respectively. We use $\langle x \rangle$ to indicate that x is 2-out-of-2 secret-shared. For 2-out-of-3 secret share, we have $x \equiv [\![x]\!]_0 + [\![x]\!]_1 + [\![x]\!]_2 \pmod{L}$, where $[\![x]\!]_k, k \in \{0, 1\}$ is sampled uniformly. The three shares are replicated over three parties as $([\![x]\!]_0, [\![x]\!]_1), ([\![x]\!]_1, [\![x]\!]_2), ([\![x]\!]_2, [\![x]\!]_0)$. So any two parties among them can reconstruct $[\![x]\!]$ and get x. The *share conversion* [30] from 2-out-of-3 to 2-out-of-2 secret share can be performed locally as $\langle x \rangle_0 \leftarrow [\![x]\!]_0 + [\![x]\!]_1 + r \pmod{L}, \langle x \rangle_1 \leftarrow [\![x]\!]_2 - r \pmod{L}$, where r is uniformly random and is synchronized between the two parties via PRG. The share conversion from 2-out-of-2 to 2-out-of-3 secret share requires one round of communication and is performed as $[\![x]\!]_0 \leftarrow -r_2 \pmod{L}, [\![x]\!]_1 \leftarrow \langle x \rangle_0 - r_0 \pmod{L}, [\![x]\!]_2 \leftarrow \langle x \rangle_1 - r_1 \pmod{L}$, where $r_0 + r_1 + r_2 \equiv 0 \pmod{L}$. See our technical report [43] for the detailed protocol of share conversion.

General MPC on Secret-Shared Data. RingSG requires a general-purpose and semi-honest secure computation protocol over 2-out-of-3 additively secret-shared data, e.g., [29]. The protocol supports common arithmetic operations (like add and multiplication) and logic operations (like comparison and two-way multiplexer [36]).

Oblivious Permutation and Aggregation. In RingSG, we utilize two special oblivious algorithms, called *Oblivious Extended Permutation (OEP)* [32] and *Oblivious Grouped Aggregation (OGA)* [41]. Intuitively, OEP is for obliviously permuting a secret-shared vector $\langle T \rangle$ according to a predefined permutation $\pi := (\text{src}, \text{dst})$. By *extended* we mean that π might include replication or absence of

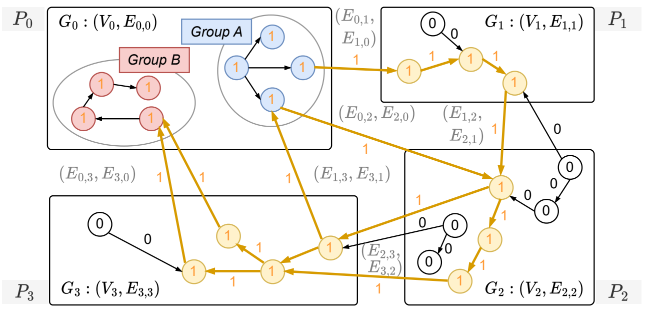


Figure 3: An example of collaborative graph processing: four-party Connected Component Labeling.

vector elements. The functionality of OEP, i.e., $\mathcal{F}_{\text{OEP}}$, is provided in Functionality 1. OGA is for aggregating elements in a secret-shared vector $\langle T \rangle$ according to a merge operation $\mathcal{F}_{\boxplus}$ and a predefined partition $\mathcal{G}$, which divides the secret-shared vector into contiguous segments. $\mathcal{F}_{\boxplus}$ is commutative and associative. The aggregation result of each segment is finally stored in the first slot of each segment. Functionality 2, i.e., $\mathcal{F}_{\text{OGA}}$, specifies OGA.

3 Problem Setting and Prior SOTA

This section introduces a cryptographic ideal functionality that formally captures the computation setting/goal and threat model of collaborative graph processing. After that, we provide a review of the SOTA approaches, and analyze their major limitations.

3.1 Collaborative Graph Processing

The collaborative graph processing involves N graph owners, denoted as $(P_0, P_1, ..., P_{N-1})$ where the indices are taken modulo N. P_i holds a local graph $G_i = (V_i, E_{i,i})$. For each edge $e \in E_{i,i}$, both e.src and e.dst are within V_i. Thus, the edges in $E_{i,i}$ are referred to as *intra-edges*. Different local graphs are interconnected by *inter-edges*. We denote the vector of edges directed from G_i to G_j as $E_{i,j}, i \neq j$. For all $e \in E_{i,j}$, both P_i and P_j are aware of (e.src, e.dst, e.data). For instance, in inter-bank transfers, both banks know the identifiers of the source account and the destination account, along with the transferred amount.

The goal of collaborative graph processing is to analyze the *global graph*, constituted by local graphs and inter-edges, using some graph algorithms, and to obtain insights unavailable from siloed local graphs. In the representative example in Figure 3, P_0 wants to detect the connections between two groups of vertices (A and B). Although both groups are within G_0, P_0 cannot detect the connections on its own, because A and B are inter-connected by complex cross-graph links, rather than simple intra-edges. This is a common money laundering strategy for financial criminals to hide the source of illegal money [39, 44]. To detect such behaviors, all four parties need to collectively analyze the global graph.

In the ideal world, we construct functionality $\mathcal{F}_{\text{RingSG}}$ to realize collaborative graph processing. $\mathcal{F}_{\text{RingSG}}$ articulates the security guarantees that we wish to achieve with real-world protocols. In the *input stage*, $\mathcal{F}_{\text{RingSG}}$ collects local graphs and inter-edges from all N parties. It also receives the graph algorithm specification, alg, which specifies the detailed Scatter-Gather operations ($\mathcal{F}_{\text{S}}$ and $\mathcal{F}_{\text{G}}$) and algorithm iterations (*maxIter*) for processing graph data.

Functionality $\{\langle V'_i \rangle\} \leftarrow \mathcal{F}_{\text{RingSG}}$ $(\{E_{i,j}\}, \{V_i\}, \text{alg})$

Input:

1. $\forall i \in [N]$, P_i sends $(\{E_{i,j}\}, \{E_{j,i}\}, V_i)$, $j \in [N]$, to $\mathcal{F}_{\text{RingSG}}$;
2. All P_i agree on a specification, alg := $(\mathcal{F}_{\text{S}}, \mathcal{F}_{\text{G}}, \mathit{maxIter})$, and send it to $\mathcal{F}_{\text{RingSG}}$. alg.$\mathit{maxIter}$ is the number of iterations.

Compute:

1. $\mathcal{F}_{\text{RingSG}}$ constructs the global graph G, where $G = (V, E)$, $V := \cup V_i$ and $E := \cup E_{i,j}, i, j \in [N]$;
2. $\mathcal{F}_{\text{RingSG}}$ processes G as required by alg, producing the updated vertex vectors $\{V'_i\}, i \in [N]$.

Output:

1. $\forall i \in [N]$, $\mathcal{F}_{\text{RingSG}}$ sends $\langle V'_i \rangle_0$ to P_i, $\langle V'_i \rangle_1$ to P_{i+1};
2. $\mathcal{F}_{\text{RingSG}}$ sends sizes of edge/vertex vectors to P_i, i.e., $\mathcal{L} := \{|E_{i,j}|, |V_i|\}, \forall i, j \in [N]$.

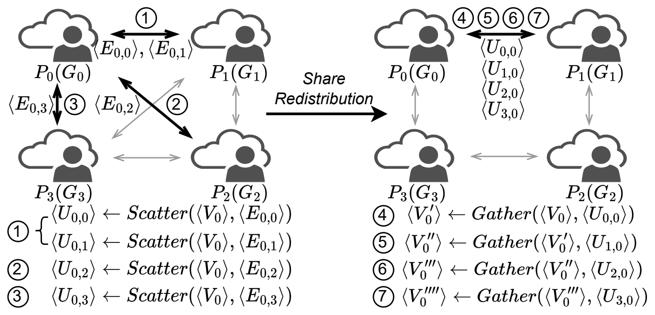
Functionality 3: $\mathcal{F}_{\text{RingSG}}$ for RingSG

In the *computation stage*, $\mathcal{F}_{\text{RingSG}}$ first concatenates the local graphs into a global graph based on inter-edges, i.e., $G = (V, E)$, $V := \cup V_i, E := \cup E_{i,j}, i, j \in [N]$. Afterwards, it executes the graph algorithm as specified in alg, using the vertex-centric abstraction. The computation result includes the updated vertex vectors, i.e., $V'_i, i \in [N]$. In the *output stage*, $\mathcal{F}_{\text{RingSG}}$ distributes the secret shares of the updated vertex vectors to specific parties. Here we suppose that V'_i is shared between P_i and P_{i+1}. Apart from that, it also distributes sizes of vertex/edge vectors, denoted as $\mathcal{L}$, to each party. The specification of $\mathcal{F}_{\text{RingSG}}$ is provided in Functionality 3.

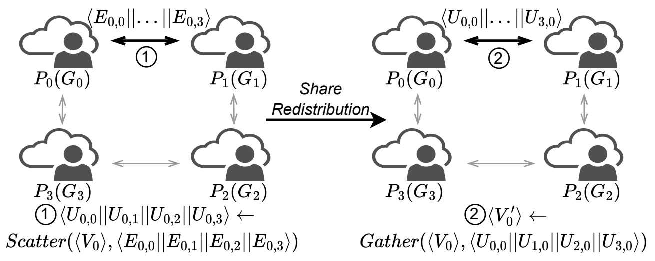
Threat Model. We consider semi-honest parties (i.e., protocol-compliant but curious) that interact with $\mathcal{F}_{\text{RingSG}}$. In addition, we assume that they do not collude with each other. Throughout the interaction with $\mathcal{F}_{\text{RingSG}}$, the information that each party P_i can observe is limited to: (i) the local graph G_i; (ii) the relevant inter-edges $E_{i,j}, E_{j,i}, j \in [N] \backslash \{i\}$; (iii) the sizes of vertex/edge vectors. P_i can not observe other parties' local graphs, irrelevant inter-edges and any data computed based on this information. The final updated vertex vectors are secret-shared to enable composition of $\mathcal{F}_{\text{RingSG}}$ with other secure computation components to form end-to-end graph processing applications (instantiations). Different from GraphSC (outsourced computation), both CoGNN and $\mathcal{F}_{\text{RingSG}}$ reveal the local graph sizes as graph owners themselves are protocol participants. In § 8, we discuss the rationale behind this threat model.

3.2 State-of-the-Art and Limitations

To support execution of various graph algorithms in a privacy-preserving manner, GraphSC [34] is the de facto standard, mainly because of its parallel efficiency, vertex-centric abstraction hiding away algorithm-specific details and strong security guarantees offered by the underlying MPC protocols. GraphSC organizes the global graph as a vector with one vertex or edge occupying one slot. The whole vector is secret-shared among several computing parties. Given this input setting, GraphSC tries to express the vertex-centric abstraction in a privacy-preserving (i.e., oblivious) way. For Scatter, GraphSC first obliviously sorts the vector to place each vertex and its outgoing edges in contiguous segments. Afterwards, it propagates vertex data to edges using an oblivious *propagate* operation on the sorted vector. For Gather, the vector is sorted again to place


①
$\langle E_{0,0}\rangle, \langle E_{0,1}\rangle$
$P_0(G_0)$
$P_1(G_1)$
$\langle E_{0,3}\rangle$ ③ $\langle E_{0,2}\rangle$ ②
Share Redistribution
$P_3(G_3)$
$P_2(G_2)$
④⑤⑥⑦
$\langle U_{0,0}\rangle$ $\langle U_{1,0}\rangle$ $\langle U_{2,0}\rangle$ $\langle U_{3,0}\rangle$
① $\langle U_{0,0}\rangle \leftarrow Scatter(\langle V_0\rangle, \langle E_{0,0}\rangle)$
$\langle U_{0,1}\rangle \leftarrow Scatter(\langle V_0\rangle, \langle E_{0,1}\rangle)$
② $\langle U_{0,2}\rangle \leftarrow Scatter(\langle V_0\rangle, \langle E_{0,2}\rangle)$
③ $\langle U_{0,3}\rangle \leftarrow Scatter(\langle V_0\rangle, \langle E_{0,3}\rangle)$
④ $\langle V_0'\rangle \leftarrow Gather(\langle V_0\rangle, \langle U_{0,0}\rangle)$
⑤ $\langle V_0''\rangle \leftarrow Gather(\langle V_0'\rangle, \langle U_{1,0}\rangle)$
⑥ $\langle V_0'''\rangle \leftarrow Gather(\langle V_0''\rangle, \langle U_{2,0}\rangle)$
⑦ $\langle V_0''''\rangle \leftarrow Gather(\langle V_0'''\rangle, \langle U_{3,0}\rangle)$

(a) The CoGNN paradigm.


$\langle E_{0,0}||\dots||E_{0,3}\rangle$
①
$P_0(G_0)$
$P_1(G_1)$
Share Redistribution
$\langle U_{0,0}||\dots||U_{3,0}\rangle$
②
$P_3(G_3)$
$P_2(G_2)$
① $\langle U_{0,0}||U_{0,1}||U_{0,2}||U_{0,3}\rangle \leftarrow$
$Scatter(\langle V_0\rangle, \langle E_{0,0}||E_{0,1}||E_{0,2}||E_{0,3}\rangle)$
② $\langle V_0'\rangle \leftarrow$
$Gather(\langle V_0\rangle, \langle U_{0,0}||U_{1,0}||U_{2,0}||U_{3,0}\rangle)$

(b) The Ring-ScatterGather paradigm.

Figure 4: A 4-party example comparing CoGNN and the Ring-ScatterGather paradigm.

each vertex and its incoming edges in contiguous segments, which is followed by an oblivious *aggregate* operation to aggregate edge data to vertices. Since the secret-shared vectors are sorted, both *propagate* and *aggregate* can be obliviously executed in parallel with low circuit depth. The overhead of GraphSC is primarily attributed to the expensive secure/oblivious sort. Although several approaches [10, 23] propose to replace the per-iteration sorts with one-time preprocessing sort and per-iteration shuffles, the cost of one-time sort still dominates the overall protocol running time (over 70% [10]). Moreover, because each computing party has to process the global graph, the per-party overhead grows rapidly as the number of graph owners increases.

To address this problem, CoGNN [41] proposes to decompose the overall computation workload into distributed tasks such that each party only processes parts of the global graph and the oblivious sort can be converted to computationally efficient permutation. We illustrate the workflow of CoGNN using a four-party example in Figure 4a. From the perspective of V_0, CoGNN decomposes the Scatter operation into four tasks, i.e., the Scatter from V_0 to $E_{0,0}$, $E_{0,1}$, $E_{0,2}$ and $E_{0,3}$, respectively. Note that P_0 knows the order of inputs of the four tasks in clear, so the reordering operation can be performed with secure permutation instead of sort. The first two tasks are securely executed by (P_0, P_1), while the latter two tasks are assigned to (P_0, P_2) and (P_0, P_3), respectively. As a result, P_1 is only responsible for the Scatter from V_0 to $E_{0,0}$ and $E_{0,1}$, while P_2 is responsible for $E_{0,2}$ and P_3 is responsible for $E_{0,3}$. The total communication/computation complexity of the four Scatter tasks of V_0 is $O(N|V_0| + \sum_j |E_{0,j}|)$. Summing up the Scatter tasks for all V_i ($i \in [N]$), the total complexity is $O(N|V| + |E|)$.

During the Gather phase, from the perspective of V_0, it needs to Gather four update vectors, i.e., $U_{0,0}, U_{1,0}, U_{2,0}, U_{3,0}$. According to the computation scheme of the Scatter phase, $U_{0,0}, U_{1,0}$ are originally secret-shared between (P_0, P_1), while $U_{2,0}$ is shared between (P_0, P_2), and $U_{3,0}$ is shared between (P_0, P_3). For ease of Gather computation, CoGNN performs share redistribution, to make $U_{0,0}, U_{1,0}, U_{2,0}, U_{3,0}$ all secret-shared between (P_0, P_1). After that, V_0 Gathers the four update vectors one by one, resulting in $O(N|V_0|)$ complexity. The overall complexity of the Gather phase, summing up all V_i's ($i \in [N]$) Gather tasks, is $O(N|V|)$. Taken together, the overall complexity of one vertex-centric iteration is $O(N|V| + |E|)$.

The constructions of the GraphSC-based SOTA and CoGNN exhibit three key limitations:

- *Sub-optimal Complexity:* Their computation schemes are fundamentally sub-optimal. GraphSC is $O((|V| + |E|)\log(|V| + |E|))$ due to secure sort. For CoGNN, it assigns N Scatter tasks for the same $V_i, i \in [N]$. Consequently, the total complexity of Scatter tasks is factored by N. A similar limitation applies to Gather tasks. Ideally, the complexity of an optimal scheme should be linear to $(|V| + |E|)$ and independent of N. In § 4, we discuss our computation paradigm that achieves the optimal complexity.
- *Limited Concrete Efficiency:* Within the CoGNN framework, each decomposed task is securely executed via 2PC protocols. The methodology for generalizing the execution to adopt more efficient MPC protocols remains unclear. This significantly limits the concrete runtime of CoGNN. The GraphSC-based SOTA and CoGNN require tens of minutes to process a global graph of millions of edges. In § 5, we present our novel cryptographic protocols that reduce this duration by up to an order of magnitude.
- *Lack of End-to-end Instantiation:* Both the GraphSC-based SOTA and CoGNN simply leave the post-processing graph data in secret-shared form or reveal it publicly. They lack the end-to-end protocol instantiation to properly obtain application-specific results in a privacy-preserving manner. For instance, when tracing abnormal funding flow in anti-money-laundering, the parties want to selectively open critical transfer paths, instead of the naively opening whole updated global graph. In § 6, we elaborate on the end-to-end instantiation of RingSG with protocols to extract application-specific results from the updated graph.

4 The Ring-ScatterGather Paradigm

RingSG is underpinned by a novel computation paradigm, named Ring-ScatterGather, which achieves *optimal communication and computation complexity* within the vertex-centric abstraction. The crucial distinction between Ring-ScatterGather and the computation scheme of CoGNN is that Ring-ScatterGather guarantees the exclusivity of the graph data associated with different decomposed tasks to avoid redundant computation.

4.1 An Illustrative Example

We illustrate the high-level idea of Ring-ScatterGather using the four-party example in Figure 4b. The left part of Figure 4b shows the Scatter phase of a vertex-centric iteration, while the right part shows the subsequent Gather phase in the same iteration. During the Scatter phase, Ring-ScatterGather constructs four parallelizable Scatter tasks, represented as four double-headed arrows. The four

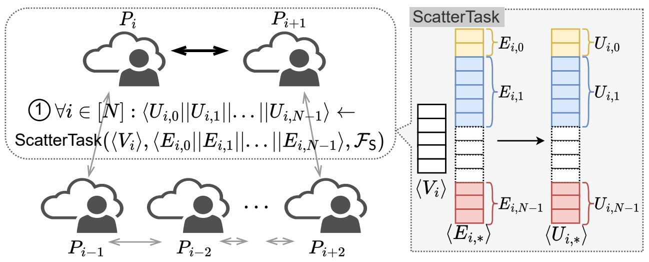

P_i
P_{i+1}
① $\forall i \in [N] : \langle U_{i,0}||U_{i,1}||\dots||U_{i,N-1}\rangle \leftarrow$
ScatterTask$(\langle V_i\rangle, \langle E_{i,0}||E_{i,1}||\dots||E_{i,N-1}\rangle, \mathcal{F}_S)$
P_{i-1}
P_{i-2}
P_{i+2}
ScatterTask
$E_{i,0}$
$E_{i,1}$
$E_{i,N-1}$
$U_{i,0}$
$U_{i,1}$
$U_{i,N-1}$
$\langle V_i\rangle$
$\langle E_{i,*}\rangle$
$\langle U_{i,*}\rangle$

Figure 5: A ScatterTask of RingSG.

tasks handle the Scatter computation for each of $V_i, i \in [4]$, and are assigned to four different groups of parties, $(P_i, P_{i+1}), i \in [4]$ accordingly for secure computation.

In Figure 4b, we zoom into the Scatter task for V_0, which is securely computed by the group (P_0, P_1). The input of this task includes $\langle V_0\rangle$ and $\langle E_{0,0}||E_{0,1}||E_{0,2}||E_{0,3}\rangle$, the vector of all edges originating from $\langle V_0\rangle$. P_0 can track the order of $\langle V_0\rangle$ and $\langle E_{0,0}||E_{0,1}||E_{0,2}||E_{0,3}\rangle$ because it owns these data (i.e., P_0 knows V_0, and all the edge vectors originating from V_0 in the clear). Meanwhile, all of these data are secret-shared to ensure that the other party P_1 in the group can not observe the private data of P_0. Similarly, the other three Scatter tasks handle the associated graph data for $\langle V_1\rangle$, $\langle V_2\rangle$ and $\langle V_3\rangle$, respectively. Therefore, the graph data processed by all four Scatter tasks are mutually exclusive. Because these tasks form a ring, we name our computation paradigm Ring-ScatterGather. The complexity of all Scatter tasks regarding V_i is $O(|V_i| + \sum_j |E_{i,j}|)$ in communication/computation. Thus, the overall Scatter tasks in this ring take $O(|V| + |E|)$. At the end of the Scatter phase, all Scatter tasks are synchronized to ensure their completion.

After the Scatter phase, Ring-ScatterGather enters the subsequent Gather phase. Similarly, the Gather phase also includes a ring of four parallel Gather tasks, each processing the Gather computation for $V_i, i \in [4]$. The four tasks are mutually exclusive, resulting in an overall complexity of $O(|V| + |E|)$. At the end of the Gather phase, Ring-ScatterGather synchronizes all parties and ensures that all Gather tasks in the ring are finished. Afterwards, Ring-ScatterGather is ready to initiate the next iteration.

The cumulative cost of an iteration comprises the aggregate cost of the Scatter and Gather phases, each of which encompasses a ring of tasks. Consequently, the overall communication/computation complexity of an iteration within RingSG is $O(|V| + |E|)$. This complexity is *optimal* because each vertex and edge has to be visited at least once in an iteration to ensure protocol obliviousness, and the complexity is irrespective of the number of parties.

In this example, each task is computed by a group of two parties in Figure 4b. In § 5, we generalize it to incorporate additional parties in each task and use more efficient MPC schemes.

4.2 Paradigm Specification

Now, we present the formal specification of Ring-ScatterGather.

Scatter. For Ring-ScatterGather in the N-party setting, during Scatter, there are N vertex vectors propagating data to their outgoing edges at the same time. Ring-ScatterGather chooses to pack all the Scatter computation relevant to one vertex vector as a separate secure computation task, denoted as ScatterTask, and assign

Protocol $\langle U\rangle \leftarrow$ ScatterTask$(\langle V\rangle, \langle E\rangle, \mathcal{F}_S)$

Input. The secret-shared vertices $\langle V\rangle$, edges $\langle E\rangle$ and $\mathcal{F}_S$.
Output. The secret-shared updates $\langle U\rangle$, where $|\langle U\rangle| = |\langle E\rangle|$.

```
1 ⟨V^src⟩ ← OEP(⟨V⟩, V.id, E.src);            # Copy vertex data to edges.
2 for i ∈ Z_{|E|}:                               # Generate a vector of updates.
3     ⟨U⟩[i] ← F_S(⟨V^src⟩[i], ⟨E⟩[i]).
```

Protocol 1: The Protocol of ScatterTask

it to one specific group of parties for execution. Figure 5 shows the ScatterTask related to $V_i, i \in [N]$. We can see that there are N outgoing edge vectors connected to V_i, i.e., $E_{i,0}, ..., E_{i,N-1}$. RingSG concatenates the N edge vectors as one edge vector $E_{i,0}||...||E_{i,N-1}$ and secret-shares it between (P_i, P_{i+1}). V_i is also shared between (P_i, P_{i+1}). It is worth noting that P_i knows V_i.id, $E_{i,j}$.src and $E_{i,j}$.dst, $j \in [N]$ in clear. Thus, we can invoke Protocol 1 to perform the ScatterTask for V_i. The protocol runs in two steps:

(1) Copy each vertex's data to the vector slots of its outgoing edges using an Oblivious Extended Permutation (OEP);
(2) Combine the copied vertex data and the edge data using $\mathcal{F}_S$ to generate an update on each edge.

In our execution of ScatterTask in Figure 5, V.id corresponds to V_i.id, while E.src corresponds to $(E_{i,0}||...||E_{i,N-1})$.src. Both vectors are known by P_i in clear but hidden from P_{i+1}.

Share Redistribution. Before discussing how Gather computation is performed, we take a look at the distribution of all the updates *targeting* V_i, among the N parties. Right after all ScatterTasks finish, there are N generated update vectors that target V_i, i.e., $U_{0,i}, ..., U_{N-1,i}$. According to the computation setting of ScatterTask, these N update vectors are secret-shared between different groups of parties, i.e., (P_0, P_1), (P_1, P_2),..., and (P_{N-1}, P_0), respectively. For the efficiency of Gather computation, we want to redistribute these shares to make the N update vectors secret-shared among a specific group of parties (P_i, P_{i+1}), who are going to perform the following Gather computation. Conversely, for the secret-shared update vector $\langle U_{i,0}||...||U_{i,N-1}\rangle$ *originating from* V_i (by ScatterTask), we need to break it into N vectors $\langle U_{i,0}\rangle, ..., \langle U_{i,N-1}\rangle$ and redistribute them to $(P_0, P_1), ..., (P_{N-1}, P_0)$, respectively:

- For $j \in [N]$, if $j \notin \{i-1, i, i+1\}$, we have P_i send $\langle U_{i,j}\rangle_0$ to P_j, and have P_{i+1} send $\langle U_{i,j}\rangle_1$ to P_{j+1}.
- Meanwhile, we have P_{i+1} send $\langle U_{i,i-1}\rangle_1$ to P_{i-1}, and have P_i send $\langle U_{i,i+1}\rangle_0$ to P_{i+2}.

After this delegation of secret shares, $U_{i,j}, i \in [N]$ is secret-shared between (P_j, P_{j+1}). Note that, before sending, we require P_i and P_{i+1} to randomize each secret-shared vector they hold independently using a synchronized PRG. For example, $\langle U_{i,j}\rangle_0 \leftarrow \langle U_{i,j}\rangle_0 - r \pmod{L}$, $\langle U_{i,j}\rangle_1 \leftarrow \langle U_{i,j}\rangle_1 + r \pmod{L}$. The secret shares each party receives during share redistribution are uniformly random, mutually independent, and also independent of their local data. They can only observe the transferred data size. Thus, the privacy guarantee is preserved.

Gather. After share redistribution, V_i and $U_{*,i} = U_{0,i}||...||U_{N-1,i}$ are secret-shared between (P_i, P_{i+1}), as shown in Figure 6. The goal of a GatherTask is to traverse the update vector $U_{*,i}$, and merge each update with their corresponding vertex in V_i. RingSG privacy-preservingly performs this in four steps, as specified in Protocol 2:

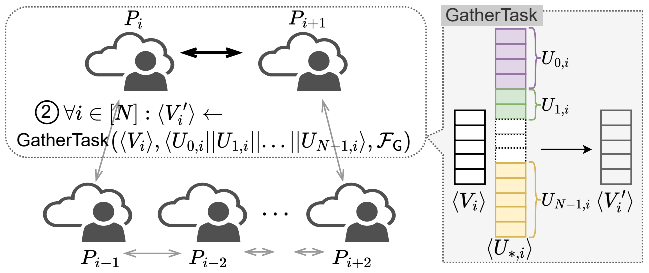

P_i
P_{i+1}
② $\forall i \in [N] : \langle V_i' \rangle \leftarrow$ GatherTask$(\langle V_i \rangle, \langle U_{0,i} || U_{1,i} || \ldots || U_{N-1,i} \rangle, \mathcal{F}_G)$
P_{i-1}
P_{i-2}
P_{i+2}
GatherTask
$U_{0,i}$
$U_{1,i}$
$U_{N-1,i}$
$\langle V_i \rangle$
$\langle V_i' \rangle$
$\langle U_{*,i} \rangle$

Figure 6: A GatherTask of RingSG.

```
Protocol ⟨V'⟩ ← GatherTask(⟨V⟩, ⟨U⟩, $\mathcal{F}_G$)
Input. The secret-shared vertices ⟨V⟩, updates ⟨U⟩ and $\mathcal{F}_G$.
Output. The secret-shared updated vertices ⟨V'⟩.
$\langle \hat{U} \rangle \leftarrow$ OEP$(\langle U \rangle, U.\text{dst}, \hat{U}.\text{dst})$;          # Sort update vector by dest.
$\langle \hat{U}' \rangle \leftarrow$ OGA$(\langle \hat{U} \rangle, \hat{U}.\text{dst}, \mathcal{F}_G)$   # Group-wise update aggregation.
$\langle \hat{U}^{\text{dst}} \rangle \leftarrow$ OEP$(\langle \hat{U}' \rangle, \hat{U}.\text{dst}, V.\text{id})$.
for $i \in \mathbb{Z}_{|V|}$:                                   # Element-wise merge operation.
    $\langle V' \rangle[i] \leftarrow \mathcal{F}_G(\langle V \rangle[i], \langle \hat{U}^{\text{dst}} \rangle[i])$.
```

Protocol 2: The Protocol of GatherTask

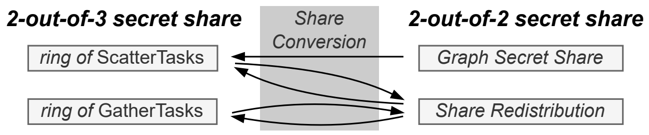


Figure 7: Share Conversion of RingSG for N-party.

(1) Obliviously reorder $U_{*,i}$ to place updates sharing the same destination vertex at adjacent slots in the vector using OEP. The permutation required by the reordering is supplied by P_i, since it knows $U_{*,i}$.dst in clear. The update vector is now turned into segments, each of which contains updates targeting the same destination;
(2) Obliviously aggregate updates sharing the same destination vertex and place the aggregation results in the first slot of each segment, via Oblivious Group Aggregation (OGA). The group information required by OGA is $U_{*,i}$.dst, also supplied by P_i. The merge operation $\mathcal{F}_{\boxplus}$ between two updates is $\mathcal{F}_G$;
(3) Obliviously extract the update aggregation results from the first slot of each segment and place them to the same slots as the corresponding vertices in V_i, using OEP;
(4) Perform element-wise merge operation (via $\mathcal{F}_G$) between the extracted update vector and the vertex vector, producing the updated vertex vector V_i'.

5 Protocols within Ring-ScatterGather

This section begins with introducing two critical protocol designs for RingSG within the Ring-ScatterGather paradigm, i.e., on-demand incorporation of 3PC and a novel OGA protocol. After that, we present the RingSG protocol that fulfills $\mathcal{F}_{\text{RingSG}}$.

5.1 On-demand Incorporation of 3PC

The Ring-ScatterGather paradigm above assumes two-party secure computation (2PC). In this section, we incorporate more efficient 3PC based on 2-out-of-3 secret share to achieve better concrete efficiency. Directly replacing 2-out-of-2 secret share with 2-out-of-3 secret share is infeasible because it violates the security model of RingSG. For example, in the share redistribution process of P_i introduced in § 4.2 (visualized in [43]), if the parties use 2-out-of-3 secret share, then P_i, P_{i+1} and P_{i+2} hold $(\llbracket U_{i,i+2} \rrbracket_0, \llbracket U_{i,i+2} \rrbracket_1)$, $(\llbracket U_{i,i+2} \rrbracket_1, \llbracket U_{i,i+2} \rrbracket_2)$ and $(\llbracket U_{i,i+2} \rrbracket_2, \llbracket U_{i,i+2} \rrbracket_0)$ respectively. If P_i delegates its shares $(\llbracket U_{i,i+2} \rrbracket_0, \llbracket U_{i,i+2} \rrbracket_1)$ to P_{i+2}, then P_{i+2} is able to reconstruct $U_{i,i+2}$, violating intermediate data privacy.

Thus, in RingSG, we propose *on-demand* usage of 3PC with replicated secret share. Specifically, as shown in Figure 7, we only switch to 2-out-of-3 secret share when executing each ScatterTask/GatherTask, and keep using 2-out-of-2 secret share during the initial graph secret share and share redistribution. Supposing that the ScatterTask/GatherTask is assigned to (P_i, P_{i+1}), right before task computation, RingSG performs share conversion from 2-out-of-2 share between (P_i, P_{i+1}) to 2-out-of-3 share among (P_i, P_{i+1}, P_{i+2}). This conversion requires transferring one piece of share between each pair of parties. Then the task is executed using 3PC. Once the task finishes, RingSG performs reversed share conversion and goes back to 2-out-of-2 secret share between (P_i, P_{i+1}). This conversion can be performed locally with synchronized PRG seeds. Then we can consistently perform share redistribution to switch between the Scatter phase and Gather phase as aforementioned under the 2-out-of-2 secret sharing setting. Essentially, in this share conversion scheme, we treat P_{i+2} as a *helper* for accelerating the Scatter/Gather tasks assigned to (P_i, P_{i+1}). Before the tasks begin, P_{i+2} receives its shares, which are immediately invalidated after the tasks finish.

With replicated secret share, the OEP functionality in RingSG is then instantiated using the three-party Oblivious Switching Network protocol (corresponding to $\mathcal{F}_{\text{SWITCH}}$) proposed in [30].

Generalization. As we explained above, the core idea of incorporating 3PC into Ring-ScatterGather is *differentiating the secret share schemes used in share redistribution and task execution.* While the share redistribution process requires 2-out-of-2 secret share, the task execution has no restrictions on the secret share scheme. Thus, Ring-ScatterGather can be generalized to use other secret share schemes (along with corresponding MPC schemes) in task execution with properly designed share conversion schemes from and to 2-out-of-2 additively secret share. The overhead shall vary according to the costs of share conversion and MPC schemes.

5.2 OGA with halved rounds

Oblivious group aggregation (OGA) is used for aggregating updates targeting the same vertices. From our experimental results in § 7.4, it takes up to 79% of the system running time and is indeed the most cost-heavy part of RingSG. The communication rounds of RingSG also mainly come from OGA, since it is logarithmic-round, while the other operations are constant-round. In particular, the SOTA OGA protocol [8, 19, 20] employs a Ladner-Fischer circuit variant, which prioritizes circuit size over depth, achieving $2\lceil\log(n)\rceil$-depth with $2n$-merge operations. Prior efforts in optimizing the SOTA's circuit depth (communication rounds) include:

- Using the original Ladner-Fischer circuit, which has $\lceil\log(n)\rceil$-depth but introduces additional $2n$ merge operations;

Protocol Oblivious Group Aggregation

Input. P_i provides $\langle T\rangle_0, \mathcal{G}$ and P_j provides $\langle T\rangle_1$, where $|\langle T\rangle| = \mathcal{G}, R := \lceil \log(|T|)\rceil$. $\langle T\rangle$ and $\mathcal{G}$ satisfy the properties specified in Theorem 1. P_i and P_j agree on the merge operation $\mathcal{F}_{\boxplus}$.
Output. P_i gets $\langle T'\rangle_0$ and P_j gets $\langle T'\rangle_1$.

```
1 $\forall r \in [R], \forall x \in \{m \cdot 2^{r+1} | m \cdot 2^{r+1} \in \mathbb{Z}_{|T|-2^r}, m \in \mathbb{Z}\}$:
2     $\langle T[x]\rangle \leftarrow \mathcal{F}_{\mathsf{MUX}_2}(C_r[x], \langle T[x]\rangle, \mathcal{F}_{\boxplus}(\langle T[x]\rangle, \langle T[x+2^r]\rangle))$,
3         $C_r[x] = (\mathcal{G}[x] \overset{?}{=} \mathcal{G}[x+2^r]) \wedge (\neg T[x+2^r].\mathsf{dummy})$.
4 $\langle T'\rangle \leftarrow \langle T\rangle$.
```

Protocol 3: The Protocol of Two-Party Oblivious Group Aggregation (Semi-honestly Secure)

- Optimizing the depth by introducing a segment tree construction [11], at the cost of $\frac{1}{2}n$ additional branching operations (OT) and increasing the memory footprint by $O(\log n)$.

In this section, we propose a new OGA protocol that yields the halved rounds (depth) without introducing extra operations or memory overhead (indicating identical communication as the SOTA). Specifically, our protocol achieves $O(n)$ communication and $(\lceil \log n\rceil + 1)\mathcal{R}$ rounds, compared to SOTA's $(2\lceil \log n\rceil - 1)\mathcal{R}$ rounds, where $\mathcal{R}$ is the communication rounds of each round of merge operation.

According to the input requirements, elements of the same group are placed in a contiguous segment of $\langle T\rangle$. Our divide-and-conquer idea is to merge every two neighboring elements $(\langle T\rangle[x], \langle T\rangle[x+1])$ in the first round $(r = 0)$ and similarly merge the results of the previous round in each subsequent round $(\langle T\rangle[x], \langle T\rangle[x+2^r])$. The key insight is that if group sizes and their order in $\langle T\rangle$ meet *specific properties*, the number of vector elements in each subsequent round can be halved. In particular, if (i) the sizes of all groups are a power of two and (ii) different groups are sorted in descending order according to their group sizes, after round r, the elements whose indices not divisible by 2^{r+1} become redundant and can be dropped, which means that the number of computed elements after each merge round is halved. Thus, we obtain an overall communication complexity of $O(|T|)$ and a round complexity of $O(\log(|T|))$.

The problem now is how to convert a normally secret-shared vector $\langle T\rangle$ into this special form. Our strategy is to obliviously insert dummy elements into $\langle T\rangle$ to round the size of each group as a power of two. Recall that before each OGA invocation in RingSG, an OEP invocation reorders the update vector, placing elements of the same group together. Right before this OEP invocation, we append dummy elements to the end of $\langle T\rangle$. The permutation π used in OEP is a composite of three subpermutations.

(1) π_0: placing elements of the same group in a contiguous segment as aforementioned;
(2) π_1: inserting the least number of dummy elements at the end of each group to round the group size as a power of two. Supposing that the original group size is d, then the rounded group size is $2^{\lceil \log(d)\rceil}$;
(3) π_2: sorting the groups in descending order, and placing unused dummy elements at the end of the permuted vector.

The three subpermutations (π_0, π_1, π_2) composing the OEP permutation $(\pi := \pi_2 \circ \pi_1 \circ \pi_0)$ are easily constructed locally by party P_i since it holds $\mathcal{G}$ in clear. For π_1, adding $(|T|-1)$ dummy elements suffices for rounding all groups (as the number of dummies per group is less than half the group size). These dummy edges, integrated into the OEP invocation before the OGA invocation, do not increase the number of rounds in RingSG, only slightly increasing OEP communication due to their inclusion in the permutation.

Our final OGA protocol is shown in Protocol 3. In particular, $\mathcal{F}_{\mathsf{MUX}_2}$ is the two-way multiplexer functionality to obliviously decide which pair of elements to merge and which not to. $C_r[x]$, the choice of $\mathcal{F}_{\mathsf{MUX}_2}$, is defined to prevent merging elements of different groups and merging dummy elements with authentic elements. C_r is computed by P_i locally. $\langle T\rangle[x].\mathsf{dummy}$ shows if $\langle T\rangle[x]$ is dummy. $\mathcal{F}_{\boxplus}$ is the operation merging the two elements. We provide an illustrative example of our OGA protocol in our technical report [43]. Notably, although Protocol 3 is specified in the two-party setting, it can be naturally migrated to the three-party setting by converting $\langle T\rangle$ to $[\![T]\!]$ and having one of the three parties hold $\mathcal{G}$.

Communication and Rounds. In merge round r, only the elements, whose indices evenly divide 2^r, are computed. So the number of involved elements is halved after each round, and the total number of conditional merge operations in line 2 is no more than $2|T|$, resulting in an overall communication complexity of $O(|T|)$.

Assuming that dummy elements have been inserted as aforementioned, i.e., before Protocol 3 is invoked, the number of merge rounds is $R := \lceil \log(|T|)\rceil = \lceil \log(|T'|)\rceil + 1$, where $|T'|$ is the vector before inserting the dummy elements.

Correctness. Since $\mathcal{F}_{\mathsf{MUX}_2}$ and the construction of C_r prevent dummy elements from merging with authentic ones, the correctness of Protocol 3 is straightforward from the following theorem:

THEOREM 1. *Suppose that $\langle T\rangle$ and $\mathcal{G}$ satisfy: (i) the number of elements in each group is a power of two and (ii) these groups are sorted in descending order of group sizes. After the merge round r of Protocol 3, the groups with sizes evenly dividing 2^{r+1} finish the aggregation, and the first element of each of these groups stores the corresponding aggregation result.*

Inductive Proof of Theorem 1. After round 0, all size-1 groups inherently store their aggregation result since they contain only a single element. For size-2 groups (whose starting indices x evenly divide $2^1 = 2$), the first element $\langle T\rangle[x]$ has successfully aggregated the value from $\langle T\rangle[x+1]$, completing the group's computation. Thus, Theorem 1 holds for $r = 0$.

Assuming Theorem 1 holds at round r, we know that after round r, groups whose sizes evenly divide 2^{r+1} have completed aggregation, with their first element storing the result. Moving to round $r+1$, the $\mathcal{F}_{\mathsf{MUX}_2}$ function in Protocol 3 (line 2) ensures that these stored results (in groups of size dividing 2^{r+1}) are not overwritten. Therefore, their first elements still hold the correct aggregation results.

Now, consider a group of size 2^{r+2} starting at element $\langle T\rangle[x]$ (where x evenly divides 2^{r+2}). This group comprises two consecutive subgroups of size 2^{r+1}: one starting at $\langle T\rangle[x]$ and the other at $\langle T\rangle[x+2^{r+1}]$. By our induction hypothesis (holding at round r), both $\langle T\rangle[x]$ and $\langle T\rangle[x+2^{r+1}]$ already store their respective subgroup aggregation results before round $r+1$ begins.

During round $r+1$, $\langle T\rangle[x]$ merges its result with $\langle T\rangle[x+2^{r+1}]$ and stores the combined result back in $\langle T\rangle[x]$. This merge operation successfully aggregates the results of the two subgroups, producing the full aggregation result for the entire group of size 2^{r+2}.

```
Protocol $\{\langle V_i' \rangle\} \leftarrow$RingSG $(\{V_i\}, \{E_{i,j}\}, \mathsf{alg})$
Input. $P_i, \forall i \in [N]$ provides $V_i, E_{i,j}, E_{j,i}, j \in [N]$. Negotiated
alg := (maxIter, $\mathcal{F}_\mathrm{S}, \mathcal{F}_\mathrm{G}$).
Output. $(P_i, P_{i+1}), \forall i \in [N]$ get $\langle V_i' \rangle$.
1  Secret-share $V_i, E_{i,*} := E_{i,0}||E_{i,1}||...||E_{i,N-1}$ between $(P_i, P_{i+1})$,
   obtaining $\langle V_i \rangle, \langle E_{i,*} \rangle$.
2  for iter ∈ [maxIter]:
3      parallel for $i \in [N]$: # ........................ Scatter
4          $\underline{(P_i, P_{i+1})}$:$\langle U_{i,*} \rangle \overset{P_{i+2}}{\Longleftarrow}$ ScatterTask($\langle V_i \rangle, \langle E_{i,*} \rangle, \mathcal{F}_\mathrm{S}$).
5          $\underline{(P_i, P_{i+1})}$: $\langle U_{i,0} \rangle, \langle U_{i,1} \rangle, ..., \langle U_{i,N-1} \rangle \leftarrow \langle U_{i,*} \rangle$.
6      parallel for $i \in [N]$: # ............ Share Redistribution
7          for $j \in [N] \backslash \{i-1, i, i+1\}$:
8              $P_i \xrightarrow{\langle U_{i,j} \rangle_0} P_j$; $P_{i+1} \xrightarrow{\langle U_{i,j} \rangle_1} P_{j+1}$.
9          if $i + 1 \neq i - 1$:
10             $P_{i+1} \xrightarrow{\langle U_{i,i-1} \rangle_1} P_{i-1}$; $P_i \xrightarrow{\langle U_{i,i+1} \rangle_0} P_{i+2}$.
11     parallel for $i \in [N]$: # ........................ Gather
12         $\underline{(P_i, P_{i+1})}$: $\langle U_{*,i} \rangle \leftarrow \langle U_{0,i} \rangle, \langle U_{1,i} \rangle, ..., \langle U_{N-1,i} \rangle$.
13         $\underline{(P_i, P_{i+1})}$: $\langle V_i' \rangle \overset{P_{i+2}}{\Longleftarrow}$ GatherTask($\langle V_i \rangle, \langle U_{*,i} \rangle, \mathcal{F}_\mathrm{G}$).
14         $\underline{(P_i, P_{i+1})}$: $\langle V_i \rangle \leftarrow \langle V_i' \rangle$.
```

Protocol 4: The RingSG Protocol (Semi-honestly Secure).

Since the theorem holds for groups of both sizes dividing 2^{r+1} and size 2^{r+2} after round $r+1$, we conclude that Theorem 1 holds for round $r+1$. The induction is therefore complete.

5.3 The RingSG Protocol

We now put all parts together to present the holistic RingSG protocol in Protocol 4. The inputs of RingSG include vertex/edge vectors from multiple parties and the graph algorithm specification. Specifically, $P_i, i \in [N]$ provides $V_i, E_{i,j}, E_{j,i}$, which is in accordance with the input of $\mathcal{F}_{\mathrm{RingSG}}$. The algorithm specification alg defines the maximum number of iterations maxIter, and the binary operations required by Scatter and Gather, i.e., $\mathcal{F}_\mathrm{S}$ and $\mathcal{F}_\mathrm{G}$, respectively.

Before all computation, P_i secret-shares V_i and $E_{i,*} := E_{i,0}||E_{i,1}||...||E_{i,N-1}$ to P_{i+1}. P_{i+1} knows the size of each vector, but can not observe the data inside. Each iteration of RingSG consists of three consecutive steps, i.e., the Scatter phase, the share redistribution phase and the Gather phase. The Scatter phase tries to propagate vertex data to their outgoing edges. During Scatter, there are N parallel tasks executed by N different groups of parties, i.e., $(P_i, P_{i+1}, P_{i+2}), i \in [N]$. Each ScatterTask($\langle V_i \rangle, \langle E_{i,*} \rangle, \mathcal{F}_\mathrm{S}$) in RingSG begins with a share conversion to make $\langle V_i \rangle, \langle E_{i,*} \rangle$ replicatedly secret-shared among (P_i, P_{i+1}, P_{i+2}) as $[\![V_i]\!], [\![E_{i,*}]\!]$. Then the task computation is efficiently performed using 3PC, producing $[\![U_{i,*}]\!]$ as its output. Right after that, a reversed share conversion turns $[\![U_{i,*}]\!]$ into $\langle U_{i,*} \rangle$, which is secret-shared between (P_i, P_{i+1}).

The share redistribution phase securely redistributes the secret shares of all $\langle U_{i,j} \rangle, i \in [N], j \in [N]$ among the N parties to prepare for the Gather phase. Since the GatherTask for $\langle V_j \rangle, j \in [N] \backslash \{i-1, i, i+1\}$ is handled by (P_j, P_{j+1}, P_{j+2}), while $\langle U_{i,j} \rangle$ is originally secret-shared between (P_i, P_{i+1}), we have (P_i, P_{i+1}) randomize their shares and send $\langle U_{i,j} \rangle_0$ to P_j, $\langle U_{i,j} \rangle_1$ to P_{j+1}. If $i+1 \neq i-1$, $\langle U_{i,i-1} \rangle$ is originally secret-shared between (P_i, P_{i+1}), we have P_{i+1} send $\langle U_{i,i-1} \rangle_1$ to P_{i-1}, and have P_i keep $\langle U_{i,i-1} \rangle_0$. Symmetrically, $\langle U_{i,i+1} \rangle$ is originally secret-shared between (P_i, P_{i+1}), we have P_i send $\langle U_{i,i+1} \rangle_0$ to P_{i+2}, and have P_{i+1} keep $\langle U_{i,i-1} \rangle_1$.

Before the Gather phase, the secret shares of all $\langle U_{j,i} \rangle, i \in [N], j \in [N]$ have been redistributed to (P_i, P_{i+1}), who then locally concatenate $\langle U_{j,i} \rangle$ and get $\langle U_{*,i} \rangle := \langle U_{0,i} \rangle || \langle U_{1,i} \rangle ||...|| \langle U_{N-1,i} \rangle$. During Gather, there are also N tasks parallelly executed by N different groups of parties. The execution of GatherTask($\langle V_i \rangle, \langle U_{*,i} \rangle, \mathcal{F}_\mathrm{G}$) also goes through share conversion to replicated secret share, 3PC-based computation and reversed share conversion. The updated vertex data $\langle V_i' \rangle$ is finally stored in $\langle V_i \rangle$ (secret-shared between (P_i, P_{i+1})) so that we can move to the next iteration.

Asymptotic Cost Analysis. For each iteration of RingSG under the vertex-centric abstraction, the computation/communication cost comes from three parts, i.e., ScatterTask, share redistribution and GatherTask.

Each ScatterTask/GatherTask begins with a share conversion from 2-out-of-2 secret share to 2-out-of-3 secret share, while ending with a reversed share conversion. The second conversion is local. The communication of the first conversion is linear to the secret-shared input size. When summing up all ScatterTasks in the **parallel for** loop of line 3 of Protocol 4, the total input size is $|V| + |E|$. For all GatherTasks in the **parallel for** loop of line 11, it is also $|V| + |E|$. As a result, in total, the share conversion operations of all tasks take $O(|V| + |E|)$ communication and $O(1)$ rounds.

Each ScatterTask requires one OEP and one $\mathcal{F}_\mathrm{S}$, which are all with linear communication/computation and constant rounds. Thus, the total communication/computation summing up all ScatterTasks is also $O(|V| + |E|)$, while the total rounds remain $O(1)$ since all ScatterTasks are concurrent. For each GatherTask, there are two invocations of OEP, one invocation of OGA and $\mathcal{F}_\mathrm{G}$, which are linear-communication, too. The total communication/computation cost summing up all GatherTasks is linear to $|V|+|E|$, i.e., $O(|V|+|E|)$. At the same time, since OGA has logarithmic rounds and its total input size is bounded by $|E|$, GatherTasks take $O(\log(|E|))$ rounds.

The share redistribution beginning from line 6 has constant rounds and the total size of secret shares transferred among all parties is less than $2\sum_{i,j} |\langle U_{i,j} \rangle| = 2|E|$. Now we can conclude that an iteration of vertex-centric computation in RingSG is $O(|V| + |E|)$ in all parties' communication/computation, and $O(\log(|E|))$ in rounds.

Detailed Cost Analysis. In [43], we provide the detailed cost analysis of RingSG ($O(|V| + |E|)$), along with the comparison with CoGNN ($O(N|V| + |E|)$), to further prove that RingSG is more efficient at computing sparse global graphs with more graph owners.

Security Theorem and Proof. Theorem 2 establishes RingSG's security in the hybrid model, assuming secure realizations of the ScatterTask and GatherTask functionalities ($\mathcal{F}_{\mathrm{ST}}$ and $\mathcal{F}_{\mathrm{GT}}$). The security of ScatterTask follows directly from its construction using OEP and $\mathcal{F}_\mathrm{S}$. For GatherTask, we prove a dedicated security lemma for the OGA protocol. The full proof of Theorem 2 via hybrid distribution construction appears in our technical report [43].

Theorem 2. *Protocol 4 securely realizes Functionality 3 in the ($\mathcal{F}_{ST}$, $\mathcal{F}_{GT}$)-hybrid model against a semi-honest, non-uniform adversary $\mathcal{A}$ corrupting one party at a time. Formally, for every PPT, semi-honest and non-uniform adversary $\mathcal{A}$ that corrupts one party P_i ($i \in [N]$), there exists a PPT, non-uniform simulator $\mathcal{S}$ corrupting the same party* *in the ideal world of $\mathcal{F}_{RingSG}$, which satisfies:*

$$REAL^{\mathcal{F}_{ST},\mathcal{F}_{GT}}_{RingSG,\mathcal{A}}(\kappa, \{E_{i,j}\}, \{V_i\}, alg)$$

$$\overset{c}{\equiv} IDEAL_{\mathcal{F}_{RingSG},\mathcal{S}}(\kappa, \{E_{i,j}\}, \{V_i\}, alg),$$

where $\mathsf{REAL}^{\mathcal{F}_{ST},\mathcal{F}_{GT}}_{\mathsf{RingSG},\mathcal{A}}(\kappa, \{E_{i,j}\}, \{V_i\}, \mathsf{alg})$ represents a joint distribution over the view of the adversary (the corrupted party's input, randomness, protocol transcript) and the protocol output, when P_i and $P_j, \forall j \in [N]\backslash\{i\}$ interact in the $(\mathcal{F}_{ST}, \mathcal{F}_{GT})$-hybrid RingSG protocol on inputs $(\{E_{i,j}\}, \{V_i\}, \mathsf{alg}), i, j \in [N]$ and computational security parameter κ; $\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{RingSG}},\mathcal{S}}(\kappa, \{E_i\}, \{E_{i,j}\}, \{V_i\}, \mathsf{alg})$ represents a joint distribution over the simulated view of the corrupted party and the functionality output, when P_i and $P_j, \forall j \in [N]\backslash\{i\}$ interact in $\mathcal{F}_{\mathsf{RingSG}}$ on inputs $(\{E_{i,j}\}, \{V_i\}, \mathsf{alg}), i, j \in [N]$ and computational security parameter κ; and $\overset{c}{\equiv}$ means the two distributions are computationally indistinguishable (in κ).

Lemma 3. *Protocol 3 securely realizes the Functionality 2 in the $(\mathcal{F}_{MUX_2}, \mathcal{F}_{\boxplus})$-hybrid model against a semi-honest, non-uniform adversary $\mathcal{A}$ corrupting either P_i or P_j. Formally, for every PPT, semi-honest and non-uniform adversary $\mathcal{A}$ that corrupts either P_i or P_j, there exists a PPT, non-uniform simulator $\mathcal{S}$ corrupting the same party in the ideal world of $\mathcal{F}_{OGA}$, which satisfies:*

$$REAL^{\mathcal{F}_{MUX_2},\mathcal{F}_{\boxplus}}_{OGA,\mathcal{A}}(\kappa, \langle T\rangle, \{C_k\}) \overset{c}{\equiv} IDEAL_{\mathcal{F}_{OGA},\mathcal{S}}(\kappa, \langle T\rangle, \{C_k\})$$

Proof Sketch for Lemma 3. For corrupted parties P_i or P_j, we define simulators $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively. Each corrupted party only observes: (i) the slots it participates in each round and (ii) inputs/transcripts/outputs of merge operations. Slot participation depends solely on round r and vector length, making it data-oblivious and easily simulated. Merge computations are simulated using $\mathcal{F}$MUX and $\mathcal{F}$merge simulators. Thus, $\mathcal{S}_i$ and $\mathcal{S}_j$ produce views indistinguishable from real executions.

6 End-to-end System Instantiation

The "bare-metal" RingSG protocol produces an updated, secret-shared global graph. To obtain certain application-specific outputs from the graph, RingSG needs to be instantiated into end-to-end application-aware systems. This section explores two anti-money laundering applications: (i) Detecting close inter-bank connections between groups of accounts; (ii) Revealing critical cross-bank transfer chains between suspicious accounts.

In application instantiation, we focus on the form of vertex/edge data, data initialization, the $\mathcal{F}_S/\mathcal{F}_G$ computation, and application-specific result extraction with minimal privacy leakage. To show how RingSG's design uniquely enables efficient result extraction, we discuss the challenges of applying similar techniques to existing SOTA methods in § 8.

6.1 Group Connection Detection

In this application, we apply the Connected Component Labeling (CC) algorithm with RingSG to detect if there is a funding flow from vertex group A to vertex group B. Without loss of generality, we assume that all vertices of A are located in G_i and all vertices of B are in G_j, where $i, j \in [N]$ and can be different or the same. The goal is to obtain a boolean flag indicating whether the two groups are connected or not, within a specific number of transfers.

Application Detect Group Connection

Objective. Detect if there is a fund flow from group $A \subseteq V_i$ to group $B \subseteq V_j$.

Input. $(V_i, E_{i,i}), i \in [N]$ (accounts and intra-bank transfers of bank P_i). $E_{i,j}, i, j \in [N], i \neq i$ (inter-bank transfers from bank P_i to P_j). v.id $\in \mathbb{Z}_{2^l}$ (account ID), v.data $\in \mathbb{Z}_2$ (account flag, indicating connected or not), e.data $\in \emptyset$.
Func. alg_{CC}: maxIter $\in \mathbb{Z}_+$ (a limit on the number of transfers), $\mathcal{F}_S\ (x, y) := x$, $\mathcal{F}_G\ (x, y) := x \vee y$.
Output. $c \in \mathbb{Z}_2$ (flag, indicating connected or not).

```
1 Input Initialization. For $v \in A \subseteq V_i$, $P_i$ sets $v$.data $\leftarrow 1$. For all the other vertices in $G$, $v$.data $\leftarrow 0$.
2 Protocol Invocation.
  $\{\langle V_i' \rangle\} \leftarrow$ RingSG($\{V_i\}, \{E_{i,j}\}, \mathsf{alg}_{CC}$), $i, j \in [N]$.
3 Result Extraction.
  (i) $(P_j, P_{j+1})$ execute $\langle V_j' \rangle \leftarrow$ OEP($\langle V_j' \rangle, V_j$.id, $\pi$.dst) to reorder $v \in B$ to the beginning of $\langle V_j' \rangle$.
  (ii) $(P_j, P_{j+1})$ execute $\langle V_j' \rangle \leftarrow$ OGA($\langle V_j' \rangle, \mathcal{G}, \mathcal{F}_G$), where $\mathcal{G}[x] = 0$ if $x \in \mathbb{Z}_{|B|}$. Otherwise $\mathcal{G}[x] = 1$.
  (iii) $(P_j, P_{j+1})$ set $\langle c \rangle \leftarrow \langle V_j' \rangle[0]$ and reconstruct $c$.
```

Application 1: Detect Group Connection

The *Input* and *Output* of Application 1 specify the exact data stored in the vertex and edge vectors. Here we set v.data over the ring $\mathbb{Z}_2$ for efficiency. The corresponding secret sharing for v.data is also over $\mathbb{Z}_2$. The *Func* details alg with the definition of maxIter, $\mathcal{F}_S$ and $\mathcal{F}_G$. During input initialization, we assign 1 to each vertex in group A to indicate connected, while assigning 0 to all the other vertices in the global graph, indicating unconnected. The protocol invocation phase executes the CC algorithm to propagate the labels across the global graph. The protocol outputs are secret-shared, representing the updated labels stored in each vertex. Finally, the result extraction phase aims to aggregate the flags (vertex data) of all the vertices in G_i, to finally output a flag indicating if there is a flow from group A to group B. The result extraction has three steps: (i) reorder the $\langle V_j' \rangle$ secret-shared between (P_j, P_{j+1}) to place vertices in group B at the beginning of $\langle V_j' \rangle$, with OEP; (ii) aggregate the vertex data in group B using OGA, by assigning group identifier 1 to vertices in B, and assigning group identifier 0 to all the other vertices; (iii) the first element in group B, i.e., $\langle V_j' \rangle[0]$ stores the aggregate flag that can be reconstructed as the application output.

6.2 Trace Transfer Chain

Besides detecting a connection between two suspicious account groups, it is also crucial to trace the core transfer paths between them in anti-money laundering efforts.

Application 2 employs a heuristic approach to discover and extract the most important transfer paths from group A to B. In this application, we utilize the Shortest Path algorithm and derive the distance of each edge using the reciprocal of the corresponding transferred amount. The smaller the distance is, the larger the amount of money is transferred via this edge. As a result, the shortest path tends to reflect an important transfer chain. Compared to

Application Trace Transfer Chain

Objective. Reveal the most important transfer paths from group $A \subseteq V_i$ to group $B \subseteq V_j$.

Input. $(V_i, E_{i,i}), i \in [N]$ (accounts and intra-bank transfers of P_i). $E_{i,j}, i, j \in [N], i \neq i$ (inter-bank transfers from P_i to P_j). $v.\mathsf{id} \in \mathbb{Z}_{2^l}$ (account ID), $v.\mathsf{data} := (\mathsf{flag}, \mathsf{id}, \mathsf{dist}) \in \mathbb{Z}_2 || \mathbb{Z}_{2^l} || \mathbb{Z}_{2^l}$ (account data, (connection flag, ID of predecessor, minimal distance)), $e.\mathsf{data} \in \mathbb{Z}_{2^l}$ (distance).
Func. alg_{SP}: $\mathsf{maxIter} \in \mathbb{Z}_+$ (a limit on the number of transfers), $\mathcal{F}_\mathrm{S}$ $((x_0, y_0, z_0), z_1) := (x_0, y_0, z_0 + z_1)$, $\mathcal{F}_\mathrm{G}$ $((x_0, y_0, z_0), (x_1, y_1, z_1)) := (x_0 \vee x_1, y_0, z_0)$ if $z_0 < z_1$. Otherwise $:= (x_0 \vee x_1, y_1, z_1)$.
Output. $\{v | v \in V \wedge v.\mathsf{data.flag} = 1\}$ (connected vertices).

1 *Input Initialization.* For $v \in A \subseteq V_i$, P_i sets $v.\mathsf{data} = (1, \perp, 0)$. For all the other vertices in G, $v.\mathsf{data} \leftarrow (0, \perp, \infty)$. $\{\widetilde{V_i} | i \in [N]\}$ copies $\{V_i | i \in [N]\}$, but only sets $v \in B \subseteq V_j$ to $(1, \perp, 0)$, while setting other vertices to $(0, \perp, \infty)$.
2 *Protocol Invocation.*
(i) $\{\langle V_i' \rangle\} \leftarrow \mathsf{RingSG}(\{V_i\}, \{E_{i,j}\}, \mathsf{alg}_{SP}), i, j \in [N]$.
(ii) $\{\langle \widetilde{V_i}' \rangle\} \leftarrow \mathsf{RingSG}(\{\widetilde{V_i}\}, \{\widetilde{E_{i,j}}\}, \mathsf{alg}_{SP}), i, j \in [N]$. $\{\widetilde{E_{i,j}}\}$ represents reversed $\{E_{i,j}\}$.
3 *Result Extraction.*
(i) Each pair of parties $(P_i, P_{i+1}), i \in [N]$ sets $\langle V_i'.\mathsf{data.flag} \rangle$ to $\langle V_i'.\mathsf{data.flag} \rangle \wedge \langle \widetilde{V_i}'.\mathsf{data.flag} \rangle$. Then, they perform $\langle V_i'' \rangle \leftarrow \mathsf{Shuffle}(\langle V_i' \rangle)$ using two OEPs. $\langle V_i''.\mathsf{data.dist} \rangle \leftarrow \langle V_i'.\mathsf{data.dist} \rangle || \langle \widetilde{V_i}'.\mathsf{data.dist} \rangle$.
(ii) $(P_i, P_{i+1}) \in [N]$ reconstruct $V_i''.\mathsf{data.flag}$ first. Then reconstruct $\{v | v \in V_i'' \wedge v.\mathsf{data.flag} = 1\}$.

Application 2: Trace Transfer Chain

Application 1, the vertex data of Application 2 is more complex. v.data stores a tuple, indicating (i) whether the vertex is connected; (ii) the predecessor of this vertex that contributes to its shortest path; (iii) the distance of this vertex. Meanwhile, e.data represents the distance of the edge. $\mathcal{F}_\mathrm{S}$ adds up the vertex's distance with the edge's distance to create an update for the destination vertex of the edge. $\mathcal{F}_\mathrm{G}$ decides whether to relax the current distance using an update.

The input initialization phase creates two sets of vertex vectors, i.e., $\{V_i | i \in [N]\}$ and $\{\widetilde{V_i} | i \in [N]\}$, where the former one initializes A as connected, but the latter one initializes B as connected. The protocol execution goes in two reversed directions at the same time, based on $\{V_i | i \in [N]\}$ and $\{\widetilde{V_i} | i \in [N]\}$ respectively. One direction finds out the shortest paths from A to the other vertices within maxIter transfers. The opposite direction finds out the shortest paths from B to the other vertices. During result extraction, we select out the vertices in the paths from group A to B by combining the connection flags produced by these two directions. The selection requires an element-wise AND ($\wedge$) and a shuffle, which includes two OEPs. Both operations have linear complexity. Finally, we reconstruct these selected vertices, which constitute the major transfer paths from group A to B with a high possibility.

7 Evaluation

Focusing on our core contributions, this section extensively evaluates RingSG from the following aspects:

Q1 The efficiency of the Ring-ScatterGather paradigm: How is the Ring-ScatterGather paradigm compared to prior SOTAs in various settings with different sizes of graphs, numbers of parties and average vertex degrees?

Q2 The efficiency of on-demand incorporation of 3PC: Does combining 2-out-of-2 secret share and 2-out-of-3 secret share via share conversion bring better concrete efficiency?

Q3 The efficiency of the OGA protocol: What is the breakdown of RingSG's running time, and does the new OGA design really contribute to a lower overall system running time?

Q4 The efficiency of application-specific result extraction: In the two end-to-end instantiations of RingSG, what is the additional cost of application-specific result extraction?

7.1 Implementation & Setup

7.1.1 Implementation. We implement a prototype of RingSG in about 3000 lines of C++ code[1]. The 3PC backend of RingSG is ABY3 [29], which also realizes the three-party OEP protocol proposed in [30]. To provide a comprehensive performance evaluation of RingSG, we implement the Ring-ScatterGather paradigm and instantiate three canonical graph algorithms within it: Connected Component Labeling (CC), Shortest Path (SP), and PageRank (PR). These algorithms exhibit diverse computational patterns within the Scatter-Gather abstraction, resulting in varying overheads across different vertex-centric phases. We focus on traditional graph processing workloads rather than Graph Neural Network (GNN) training/inference, as the latter is primarily dominated by non-graph operations like secure matrix multiplication, which fall outside the primary scope of RingSG.

To demonstrate the efficiency of our system, we employ two SOTA approaches for comparison. First, we implement the SOTA GraphSC approach [23] based on ABY3 (the same MPC scheme as RingSG). Note that we exclude the cost of secure sort in this implementation, meaning the actual communication and computation overhead of this baseline approach would be even higher. Second, we implement the CoGNN [41] approach that incorporates 3PC based on our share conversion mechanism. This implementation achieves performance that is orders of magnitude faster than the original 2PC prototype described in their paper [42]. Additionally, in § 7.3, we present an ablation study comparing our system with the original CoGNN implementation to specifically illustrate the efficiency improvements achieved solely through the incorporation of 3PC when executing the decomposed MPC-based graph processing tasks.

7.1.2 Setup. Our evaluation environment is a Linux server equipped with a multi-core x86_64 Intel CPU at 2.60GHz. The network environments, i.e., bandwidth and latency of multiple parties, are simulated using the tc command and network namespace provided by Linux. The LAN environment is of ($4000Mbps, 1ms$), while the WAN environment is of ($200Mbps, 10ms$). At the same time, since different schemes run in different multi-threading patterns, for the

[1]https://github.com/CBackyx/RingSG/tree/dev-graph

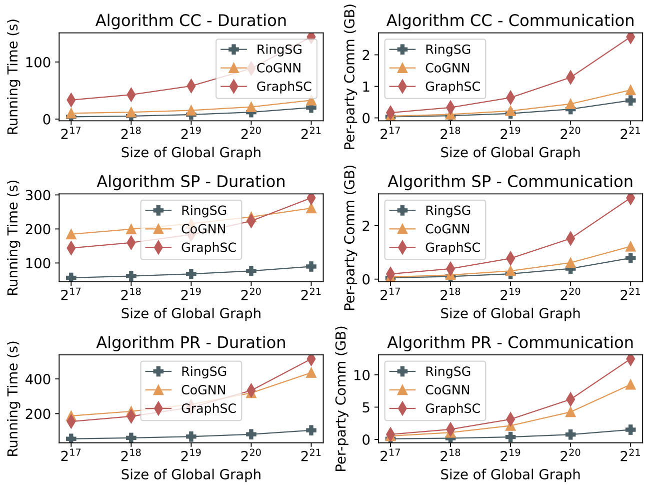


Figure 8: Running time and communication for different global graph sizes (in LAN, 8 parties, 5 iterations).

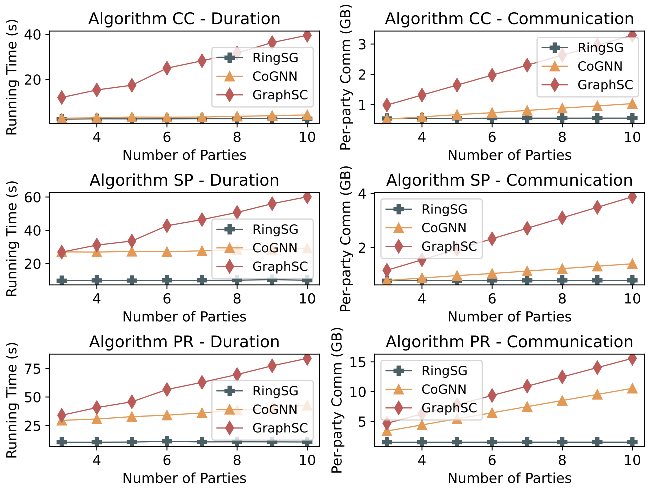


Figure 9: Running time and communication for different numbers of parties (in LAN, local graph of 2^{16}, 5 iterations).

fairness of comparison, we use the taskset –cpu-list command to limit the computing resource each scheme can use.

The obliviousness of all compared schemes guarantees that the running duration and communication depend only on the size of graphs, the number of parties and the average vertex degree, regardless of actual vertex/edge data or the ratio of inter-edges among all edges (proved by detailed cost analysis in [43]). Thus, we vary these three hyperparameters to fully evaluate RingSG under a wide range of settings (each with a corresponding multi-party global graph). The vertices are evenly distributed among all parties. By default, we set the average vertex incoming/outgoing degree to 3, the ratio of inter-edges among all edges to 0.4, and the number of parties to 8. In this default setting, supposing that each local graph has $2^x (x \in \mathbb{Z}_+)$ vertices, the size of the global graph, i.e., $|V| + |E|$, equals $4N \cdot 2^x$, where N is the number of parties. We set the ring of our two secret share schemes to $\mathbb{Z}_{2^{64}}$. Each experiment is run for five times and we take the average of the running durations.

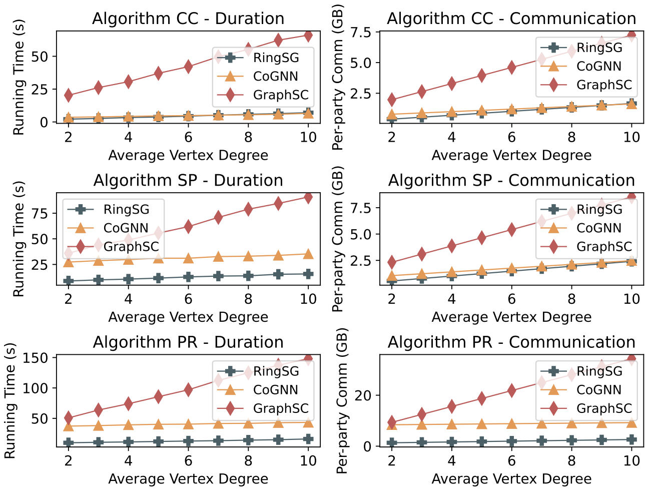


Figure 10: Running time and communication for different average vertex degrees (in LAN, 8-parties, 5 iterations).

7.2 Q1: Efficient Ring-ScatterGather Paradigm

7.2.1 Different sizes of global graphs. Figure 8 shows the running time (duration) and per-party communication of different schemes for different global graph sizes (from 2^{17} to 2^{21}). This experiment is performed in a LAN environment with 8 parties. The three rows of subfigures correspond to three different graph algorithms. Each graph algorithm is run for 5 iterations. Compared to GraphSC, when the graph size is 2^{17}, RingSG reduces the running time by $2.55 \sim 8.48\times$. When the graph size is 2^{21}, RingSG reduces the running time by $3.25 \sim 7.17\times$. As for the per-party communication, RingSG reduces GraphSC's communication by $3.92 \sim 8.28\times$ when the graph size is 2^{17}, and by $3.85 \sim 8.29\times$ when the graph size is 2^{21}. The reduction ratio of both the running time and per-party communication is stable as the graph size increases, since the cost of GraphSC without secure sort is also linear to $|V| + |E|$. When compared to CoGNN, we can see that the reduction ratio of per-party communication is stable for different graph sizes, ranging between $1.55 \sim 5.66\times$. This is in conformity with our complexity analysis of CoGNN and RingSG, which indicates that when the number of parties and average vertex degree are fixed, the per-party communication ratio between the two schemes is also fixed. The running time reduction ratio of RingSG with respect to CoGNN is $2.59 \sim 3.39\times$ for 2^{17}, and $1.64 \sim 4.19\times$ for 2^{21}.

7.2.2 Different numbers of parties. Figure 9 shows the duration and communication overhead of executing the three algorithms under different schemes and with different numbers of parties (from 3 to 10). We set a LAN environment and set the number of vertices in each local graph to 2^{16}. The reduction ratios of running time/communication of RingSG with respect to both GraphSC and CoGNN are enlarged as the number of parties increases, which confirms our cost analysis and shows RingSG's superior scalability. In particular, when there are 6 parties, compared to GraphSC, RingSG reduces the communication of GraphSC by $2.95 \sim 6.22\times$. When there are 10 parties, it is $4.91 \sim 10.36\times$. The communication reduction ratio of RingSG with respect to CoGNN is $1.33 \sim 4.29\times$ for 6 parties, and is $1.86 \sim 7.01\times$ for 10 parties. As for the running

	GraphSC	CoGNN-2PC	CoGNN	**RingSG**
CC	4.51 (0.47)	172.67 (2.11)	0.71 (0.18)	**0.49 (0.11)**
SP	7.98 (0.57)	172.32 (2.14)	5.77 (0.25)	**1.97 (0.16)**
PR	11.7 (2.48)	216.22 (3.45)	7.65 (1.74)	**2.13 (0.31)**

Table 2: Duration [s] and per-party communication [GB, in bracket (*)] of various schemes (in LAN, 8 parties, local graph of 2^{16}, amortized across 20 iterations). Note that CoGNN incorporates 3PC via our share conversion mechanism, while CoGNN-2PC is the original 2PC-based implementation.

time, RingSG reduces GraphSC's by 4.32 ~ 9.94× for 6 parties, and 6.04 ~ 15.34× for 10 parties. RingSG reduces CoGNN's by 1.28 ~ 3.04× for 6 parties, and 1.65 ~ 4.01× for 10 parties.

7.2.3 Different average vertex degrees. Figure 10 shows the duration/communication overhead of different schemes for an increasing average vertex degree (from 2 to 10). Performed in a LAN environment, the experiment sets the number of parties to 8 and the number of vertices in each local graph to 2^{16}. The duration/communication reduction of RingSG compared to GraphSC increases with growing average vertex degree, whereas the reduction compared to CoGNN diminishes. For SP and PR, RingSG's per-party duration/communication stays lower than CoGNN's for all tested settings. For CC, they have nearly identical duration/communication when the average degree is 8 or more, and CoGNN appears to be more efficient than RingSG as the degree grows.

In summary, the evaluation results, consistent with our detailed cost analysis in the technical report [43], show that compared to prior SOTAs, RingSG is more efficient in processing a large and sparse global graph with more graph owners.

7.3 Q2: Efficient On-demand Incorporation of 3PC

To demonstrate the efficiency gains achieved through the on-demand incorporation of 3PC, we compare the per-iteration costs across four different schemes. The compared schemes include 3PC-based GraphSC (GraphSC), the original 2PC-based CoGNN (CoGNN-2PC), our re-implementation of CoGNN that incorporates 3PC via our on-demand share conversion (CoGNN), and our system RingSG. The results are summarized in Table 2, Note that we exclude the cost of secure sort operations in GraphSC, indicating that its actual total cost would be substantially higher. The substantial performance gap between CoGNN-2PC and CoGNN demonstrates that, despite the additional communication overhead introduced by share conversion, the incorporation of three-party computation significantly improves the efficiency of both CoGNN and our system. Specifically, across the three evaluated algorithms, the on-demand incorporation of 3PC achieves up to 243× reduction in execution time and up to 11× reduction in communication overhead.

7.4 Q3: Efficient OGA Protocol

In Table 3 and Table 4, we break down the running time of RingSG and CoGNN into the Scatter and Gather phases. Additionally, we measure the running time taken by the oblivious group aggregation (OGA) in these two schemes. Note that in RingSG, OGA is performed in Gather, while in CoGNN it is performed in Scatter.

	RingSG Scatter	RingSG Gather (OGA)	**RingSG Total**	CoGNN Scatter (OGA)	CoGNN Gather	CoGNN Total
CC	0.07	**0.40 (0.24)**	**0.47**	0.54 (0.33)	0.13	0.67
SP	0.18	**1.75 (1.53)**	**1.93**	4.81 (4.40)	0.78	5.59
PR	0.42	**1.72 (1.44)**	**2.14**	6.79 (4.27)	0.79	7.58

Table 3: Durations [s] in LAN (global graph of 2^{21}, 8 parties).

	RingSG Scatter	RingSG Gather (OGA)	**RingSG Total**	CoGNN Scatter (OGA)	CoGNN Gather	CoGNN Total
CC	0.65	**3.13 (1.76)**	**3.78**	4.85 (2.33)	1.52	6.37
SP	1.79	**15.91 (13.79)**	**17.60**	45.40 (41.25)	6.64	52.04
PR	5.45	**15.28 (13.15)**	**20.73**	79.93 (39.57)	6.69	86.62

Table 4: Durations [s] in WAN (global graph of 2^{21}, 8 parties).

	Scheme	**RingSG**	CoGNN	GraphSC
Detect Group Connection	Protocol Invocation	**34.34 (4.55)**	38.57 (14.52)	450.74 (39.40)
	Result Extraction	**0.46 (0.02)**	/	/
	Total	**34.80 (4.57)**	/	/
Trace Transfer Chain	Protocol Invocation	**56.30 (6.47)**	102.72 (20.03)	549.18 (47.40)
	Result Extraction	**1.65 (0.28)**	/	/
	Total	**57.95 (6.75)**	/	/

Table 5: Duration [s] and per-party communication [GB, in bracket (*)] of RingSG instantiations (In LAN, 8 parties, global graph of 2^{24}). Prior works lack result extraction.

Table 3 and Table 4 are run in LAN and WAN respectively. The number of parties is 8, while the local graph size is 2^{16}. We can see that, in LAN, OGA takes up 51% ~ 79% of the running time of RingSG, and 49% ~ 79% of that of CoGNN. In WAN, OGA takes up 47% ~ 78% of the running time of RingSG, and 37% ~ 79% of that of CoGNN. Thus, in both LAN and WAN, OGA contributes to an important part of the overall protocol running time, and therefore optimizing the construction of OGA is critical. Compared with CoGNN, the OGA protocol in RingSG is highly efficient, reducing the running time by 1.37 ~ 2.97× in the LAN environment and 1.32 ~ 3.01× in the WAN environment.

Our technical report [43] provides a step-wise inspection of RingSG costs to better compare OGA with other critical operations.

7.5 Q4: Efficient App-Specific Result Extraction

We measure the running time/communication of the two end-to-end instantiations of RingSG (introduced in §6) under the setting of 8 parties, LAN, and a global graph of size $2^{24} = 16,777,216$. The number of iterations during protocol invocation is set to 10.

For comparison, we also run the two corresponding algorithms (CC and SP) with CoGNN and GraphSC, which lack end-to-end result extraction, for the same number of iterations. The evaluation results are summarized in Table 5. For the two applications, the result extraction design in RingSG accounts for only 1.3% ∼ 2.9% of the overall running time, and 0.4% ∼ 4.2% of the overall communication. Notably, the total running time/communication of these two RingSG instantiations are both less than that of CoGNN and GraphSC, which only have protocol invocation and lack end-to-end result extraction.

8 Discussion

Reveal Local Graph Sizes. RingSG and CoGNN address the same collaborative graph processing paradigm, where graph owners perform the computational tasks directly. In contrast, GraphSC was initially designed for outsourced computation scenarios. While GraphSC can be adapted to the collaborative setting by distributing secret shares of the global graph among graph owners serving as computing parties, this adaptation necessarily reveals local graph sizes during the secret sharing process. Consequently, all three approaches provide equivalent privacy guarantees in practice.

Generalize to partially known inter-edges. RingSG can be generalized to cases when inter-edges are partially known by either of the two parties. Specifically, the necessary information for the source party of inter-edges is only the source-vertex identifiers, while the destination party must know the destination-vertex identifiers. Other inter-edge data (e.g., weights/distances) can be partially known and computed via secret sharing. This extends our problem setting of collaborative graph processing in § 3.1.

The Security Model. Unlike outsourced computation models, collaborative graph processing requires data owners to perform computations directly rather than relying on third-party servers. Consequently, collusion harms their own privacy, while malicious behavior undermines mutually beneficial collaboration. Given these inherent safeguards, we adopt a non-colluding, semi-honest security model and concentrate our efforts on optimizing computational and communication efficiency.

Future enhancements to RingSG for malicious security can proceed in three phases: group-wise computations, share redistribution, and share conversion. For group-wise computations, we could employ existing maliciously secure MPC protocols. The share redistribution and conversion phases could leverage techniques from recent maliciously secure aggregation schemes [37]. To maintain cross-phase consistency of secret shares, we could integrate commitment/verification mechanisms similar to those in [25], ensuring robust integrity preservation without efficiency loss.

Generalize to Other Secret Share and MPC Schemes. RingSG requires 2-out-of-2 additively secret share (ASS) for share redistribution. Yet, the group-wise computation can be generalized to use other secret share and MPC schemes with properly designed share-conversion schemes from 2-out-of-2 ASS. The per-party overhead shall vary according to the share conversion and MPC costs.

Process Incremental/Streaming Data. A promising future direction of extending RingSG is to support the processing of streaming/dynamic graph data with incremental costs. This extension might involve two key components: (i) securely caching prior execution history as secret-shared states in each vertex, and (ii) identifying and obliviously updating the portion of the global graph affected by new data through partial execution of RingSG. However, the specifics of secure state storage and identifying the influenced graph segment depend on the graph algorithm specifications.

Difficulties of Prior Approaches in Result Extraction. Directly applying our designs to CoGNN and GraphSC to extract application-specific results can be costly. In particular, the usage of homomorphic encryption in OEP of CoGNN makes result extraction rather expensive due to high cipher-plaintext expansion rate. Meanwhile, GraphSC must process the entire global graph even when only the result of a single local graph is of interest. Thus, the novel computation paradigm and protocol constructions of RingSG are the keys to enabling efficient and secure result extraction.

9 Conclusion

This paper presents RingSG, the first collaborative graph processing system attaining the optimal communication/computation complexity for the MPC-based vertex-centric abstraction. The core of RingSG is the Ring-ScatterGather paradigm, which organizes the overall secure graph computation workload into rings of parallel and non-overlapping tasks and distributes them to different groups of parties. Within Ring-ScatterGather, we propose to incorporate 3PC and a novel OGA protocol to improve its concrete efficiency. Finally, for application-specific and privacy-preserving result extraction, we present two efficient end-to-end instantiations of RingSG. Rigorous evaluations across extensive experimental settings confirm RingSG's superiority over SOTA, especially for large, sparse global graphs with growing numbers of parties.

10 Acknowledgements

We thank the anonymous reviewers for their insightful feedback. The research is supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62472247, 62425201, 62132011 and 62221003, and the National Key R&D Program of China under Grant 2024YFB2906803 and 2022YFB2403900, as well as a CIE-Smartchip research grant. The corresponding author of this paper is Zhuotao Liu.

References

[1] 2025. Complete guide to GDPR compliance. https://gdpr.eu/ Accessed: 2025-05-01.
[2] 2025. Connected-component labeling — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Connected-component_labeling, Accessed: 2025-05-01.
[3] 2025. PageRank — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/PageRank, Accessed: 2025-05-01.
[4] 2025. Shortest path problem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Shortest_path_problem, Accessed: 2025-05-01.
[5] Abdelrahaman Aly and Sara Cleemput. 2017. An Improved Protocol for Securely Solving the Shortest Path Problem and its Application to Combinatorial Auctions. *IACR Cryptol. ePrint Arch.* 2017 (2017), 971.
[6] Abdelrahaman Aly and Sara Cleemput. 2022. A fast, practical and simple shortest path protocol for multiparty computation. In *European Symposium on Research in Computer Security*. Springer, 749–755.
[7] Abdelrahaman Aly and Mathieu Van Vyve. 2015. Securely solving classical network flow problems. In *Information Security and Cryptology-ICISC 2014: 17th International Conference, Seoul, South Korea, December 3-5, 2014, Revised Selected Papers 17*. Springer, 205–221.
[8] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel privacy-preserving shortest path algorithms. *Cryptography* 5, 4 (2021), 27.

[9] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 805–817.

[10] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 610–629.

[11] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob CN Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2022. Secure parallel computation on privately partitioned data and applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 151–164.

[12] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2022. Secure Parallel Computation on Privately Partitioned Data and Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 151–164.

[13] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*. Springer, 213–230.

[14] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-shared shuffle. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*. Springer, 342–372.

[15] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III 38*. Springer, 34–64.

[16] Steven M. D'Antuono. 2018. Combating Money Laundering and Other Forms of Illicit Finance: Regulator and Law Enforcement Perspectives on Reform. https://www.fbi.gov/news/testimony/combating-money-laundering-and-other-forms-of-illicit-finance, Accessed: 2025-05-01.

[17] Frankfurt. 2022. Enhancing cooperation in the fight against money laundering. https://www.bankingsupervision.europa.eu/press/blog/2022/html/ssm.blog220524~8e08209118.en.html, Accessed: 2025-05-01.

[18] Daniel Günther, Marco Holz, Benjamin Judkewitz, Helen Möllering, Benny Pinkas, Thomas Schneider, and Ajith Suresh. 2022. Privacy-Preserving Epidemiological Modeling on Mobile Graphs. *arXiv preprint arXiv:2206.00539* (2022).

[19] Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Koji Chida. 2023. Efficient decision tree training with new data structure for secure multi-party computation. In *Proceedings on Privacy Enhancing Technologies Symposium (PoPETs)*. 343–364.

[20] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable collaborative analytics system on private database with malicious security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1740–1753.

[21] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590.

[22] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 506–525.

[23] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4017–4031.

[24] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.* 8, 3 (nov 2014), 281–292.

[25] Hidde Lycklama, Alexander Viand, Nicolas Küchler, Christian Knabenhans, and Anwar Hithnawi. 2024. Holding secrets accountable: Auditing privacy-preserving machine learning. In *33th USENIX Security Symposium (USENIX Security 24)*. USENIX Association.

[26] 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 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146.

[27] Sahar Mazloom and S. Dov Gordon. 2018. Secure Computation with Differentially Private Access Patterns. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 490–507.

[28] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. 2020. Secure parallel computation on national scale volumes of data. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2487–2504.

[29] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35–52.

[30] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1271–1287.

[31] Payman Mohassel and Saeed Sadeghian. 2013. How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 557–574.

[32] Payman Mohassel and Saeed Sadeghian. 2013. How to hide circuits in MPC an efficient framework for private function evaluation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 557–574.

[33] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[34] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. 377–394.

[35] Benjamin Ostrovsky. 2024. Privacy-Preserving Dijkstra. In *Advances in Cryptology – CRYPTO 2024*, Leonid Reyzin and Douglas Stebila (Eds.). Springer Nature Switzerland, Cham, 74–110.

[36] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 325–342.

[37] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. 2023. Elsa: Secure aggregation for federated learning with malicious actors. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1961–1979.

[38] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. 2019. Secure multiparty PageRank algorithm for collaborative fraud detection. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer, 605–623.

[39] Connie Diaz De Teran. 2023. Collaboration Is Key in the Fight Against Anti-Money Laundering. https://www.paymentsjournal.com/collaboration-is-key-in-the-fight-against-anti-money-laundering/, Accessed: 2025-05-01.

[40] Andrew C. Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*. 160–164.

[41] Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. 2024. CoGNN: Towards Secure and Efficient Collaborative Graph Learning. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4032–4046.

[42] Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. 2024. CoGNN: Towards Secure and Efficient Collaborative Graph Learning (Artifacts). https://doi.org/10.5281/zenodo.11210094

[43] Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. 2025. RingSG: Optimal Secure Vertex-Centric Computation for Collaborative Graph Processing. Cryptology ePrint Archive, Paper 2025/1209. https://eprint.iacr.org/2025/1209

[44] David Lewis Zoë Newman, Howard Cooper. 2022. Collaboration is key: how central banks are tackling money laundering. https://www.centralbanking.com/central-banks/governance/7937741/collaboration-is-key-how-central-banks-are-tackling-money-laundering, Accessed: 2025-05-01.