HERO: A Hierarchical Set Partitioning and Join Framework for Speeding up the Set Intersection Over Graphs

BOYU YANG, School of Data Science, Fudan University, China
WEIGUO ZHENG*, School of Data Science, Fudan University, China
XIANG LIAN, Department of Computer Science, Kent State University, USA
YUZHENG CAI, School of Data Science, Fudan University, China
X. SEAN. WANG, School of Computer Science, Fudan University, China

As one of the most primitive operators in graph algorithms, such as the triangle counting, maximal clique enumeration, and subgraph listing, a set intersection operator returns common vertices between any two given sets of vertices in data graphs. It is therefore very important to accelerate the set intersection, which will benefit a bunch of tasks that take it as a built-in block. Existing works on the set intersection usually followed the merge intersection or galloping-search framework, and most optimization research focused on how to leverage the SIMD hardware instructions. In this paper, we propose a novel multi-level set intersection framework, namely *hierarchical set partitioning and join* (HERO), by using our well-designed *set intersection bitmap tree* (SIB-tree) index, which is independent of SIMD instructions and completely orthogonal to the merge intersection framework. We recursively decompose the set intersection task into small-sized subtasks and solve each subtask using bitmap and boolean AND operations. To sufficiently achieve the acceleration brought by our proposed intersection approach, we formulate a graph reordering problem, prove its NP-hardness, and then develop a heuristic algorithm to tackle this problem. Extensive experiments on real-world graphs have been conducted to confirm the efficiency and effectiveness of our HERO approach. The speedup over classic merge intersection achieves up to 188x and 176x for triangle counting and maximal clique enumeration, respectively.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**.

Additional Key Words and Phrases: Set intersection over graphs, hierarchical set partitioning and join

ACM Reference Format:
Boyu Yang, Weiguo Zheng, Xiang Lian, Yuzheng Cai, and X. Sean. Wang. 2024. HERO: A Hierarchical Set Partitioning and Join Framework for Speeding up the Set Intersection Over Graphs. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 29 (February 2024), 25 pages. https://doi.org/10.1145/3639284

1 INTRODUCTION

In a wide spectrum of real-world applications such as social network analysis [7, 25], friend recommendation [4, 15, 17], and community search/detection [9, 28, 29], a *set intersection* problem, which obtains common elements between any two sets, has been one of the most fundamental and

*Corresponding Author: zhengweiguo@fudan.edu.cn

Authors' addresses: Boyu Yang, yangby19@fudan.edu.cn, School of Data Science, Fudan University, Shanghai, China, 200433; Weiguo Zheng, zhengweiguo@fudan.edu.cn, School of Data Science, Fudan University, Shanghai, China, 200433; Xiang Lian, Department of Computer Science, Kent State University, Kent, Ohio, USA, xlian@kent.edu; Yuzheng Cai, School of Data Science, Fudan University, Shanghai, China; X. Sean. Wang, School of Computer Science, Fudan University, Shanghai, China.

ACM 2836-6573/2024/2-ART29
https://doi.org/10.1145/3639284

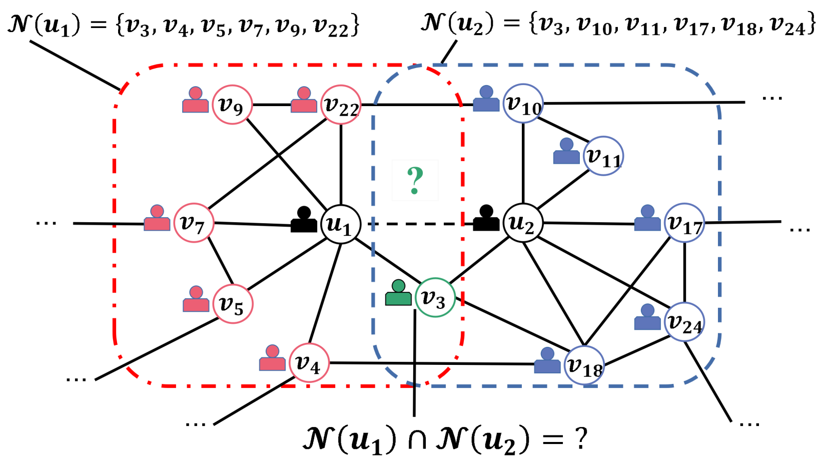

$\mathcal{N}(u_1) = \{v_3, v_4, v_5, v_7, v_9, v_{22}\}$
$\mathcal{N}(u_2) = \{v_3, v_{10}, v_{11}, v_{17}, v_{18}, v_{24}\}$
v_9
v_{22}
v_{10}
v_{11}
?
v_7
u_1
u_2
v_{17}
v_5
v_3
v_{24}
v_4
v_{18}
$\mathcal{N}(u_1) \cap \mathcal{N}(u_2) = ?$

Fig. 1. An example of the friend recommendation in a social network G.

important operators in graph algorithms/tasks such as the triangle counting [8, 27], maximal clique enumeration [5], and subgraph listing [24, 30].

1.1 Motivation Example

Below, we give a motivation example of the friend recommendation in social networks.

Example 1.1 (Friend Recommendation in Social Networks). Consider an example of social network, G, in Figure 1, where each vertex is a user and each edge between two vertices indicates the friend relationship between the two users. One important function on the social network platform (e.g., Twitter or Facebook) is to help users find and connect their friends. Intuitively, if two users have many common friends on social networks, they are more likely to know each other in reality. Therefore, in this case, we need to obtain a set of common friends between these two users and recommend them to add each other as friends (if this set is large).

Figure 1 illustrates two users u_1 and u_2 who are not connected in social network G, where their 1-hop neighbor sets are given by $\mathcal{N}(u_1) = \{v_3, v_4, v_5, v_7, v_9, v_{22}\}$ and $\mathcal{N}(u_2) = \{v_3, v_{10}, v_{11}, v_{17}, v_{18}, v_{24}\}$, respectively. To find their common friends, we need to conduct the set intersection operator, $\mathcal{N}(u_1) \cap \mathcal{N}(u_2)$, between their 1-hop neighbor sets, and obtain common friends (i.e., $\{v_3\}$ in this example). ■

Due to the large scale of social networks G in Example 1, in the worst case, there are quadratic pairs of user vertices (or set intersections) that need to be checked. Therefore, it is rather important, yet challenging, to optimize the cost of performing each set intersection operator, which inspires our work in this paper.

1.2 Existing Methods and Limitations

Existing works on the set intersection [14, 16, 23] follows a widely-used framework, the *merge intersection*, which assumes two sorted sets A and B (w.l.o.g. in the ascending order) and scans to merge both lists at the same time. The time complexity of this merge intersection method is $O(|A|+|B|)$ in the worst case, where $|A|$ and $|B|$ are the numbers of elements in A and B, respectively.

Another classic approach to perform the set intersection is the *galloping search* [10]. Let us assume that $|A| < |B|$. For each element u in A, the galloping search determines whether u is contained in B by using a binary search. The time complexity of this method is $O(|A| \log |B|)$, thereby it is more efficient when $|A| \ll |B|$ holds.

A bunch of algorithms have been developed powered by SIMD [1, 14, 16]. For example, QFilter [14] improves the merge intersection by leveraging the SIMD instructions and a proposed byte-checking filter. However, due to the variability in available SIMD support across different processor architectures, the SIMD instruction sets are architecture-specific, leading to the inconvenience of implementing algorithms. In the worst-case scenario, some processors may not have SIMD instructions at all, making the SIMD-based algorithms even unable to work. Recently, a reducing-merging framework [31] has been devised to enhance the performance of merge intersection. The basic idea is to reduce the input sets as much as possible before conducting intersection. The corresponding time cost is $O(\log |A| + \log |B| + |A'| + |B'|)$, where A' and B' refer to subsets retrieved from A and B with range code reduction, respectively. The reducing-merging framework highly depends on the reducing performance, however, it remains an open problem to maximize the reducing ability.

Although advanced techniques such as SIMD [14] and set reduction [31] have been proposed to boost the set intersection task, they all take merge intersection or galloping-search paradigms as the backbone. In contrast, we focus on the following question: ***Can we develop a novel and more powerful framework to make set intersections on graphs even faster in practice?***

1.3 Our Contributions

In order to speed up the intersection $A \cap B$ between two sets A and B, in this paper, we design a novel multi-level set intersection framework, namely *hierarchical set partitioning and join* (HERO), which consists of *offline set decomposition* and *online set intersection* stages. The intuition behind our HERO framework is to decompose the task of the set intersection into several intersection subtasks with smaller subsets and avoid the computation cost if one of the two subsets to intersect in subtasks is empty.

Specifically, in the offline set decomposition phase, we divide the universal set into disjoint partitions and project each set A (or B) onto these partitions (called *set partitioning*), leading to projected subsets. Then, we take the IDs of non-empty (projected) subsets and form a new partition ID set. We recursively project this new partition ID set into partition ID subsets. This way, we can offline build a hierarchical structure of (element or partition ID) subsets for the original set A (or B).

In the online set intersection phase, we will start from the top of the hierarchical structures for sets A and B to perform the join over partition ID sets (called *partition ID join*). Then, we can traverse hierarchical structures of A and B to the next-level (sub)set intersection, based on the partition IDs in the join results, in a recursive top-down manner.

Under the HERO framework, we also propose an effective index, namely *set intersection bitmap tree* (SIB-tree) for the set intersection in the graph. We notice that the bitwise operators, supported by most computers, enable bit-level parallel comparisons, implying a great potential to accelerate the set intersection task. To the end, it is necessary to represent each set using a bitmap where each bit denotes whether or not a vertex appears in the set. However, maintaining bitmaps for sets of neighbors for all the vertices results in the overhead $O(|V|^2)$, which is impractical for large-scale graphs. Moreover, due to the size limit of an operand register (namely word length), the bitwise operators (such as boolean AND operation) on over-width bitmaps cannot be completed by a single instruction. To handle the problems, we construct the SIB-tree, that is, plugging the bitwise operators into the HERO framework by decomposing the original set intersection into small-sized

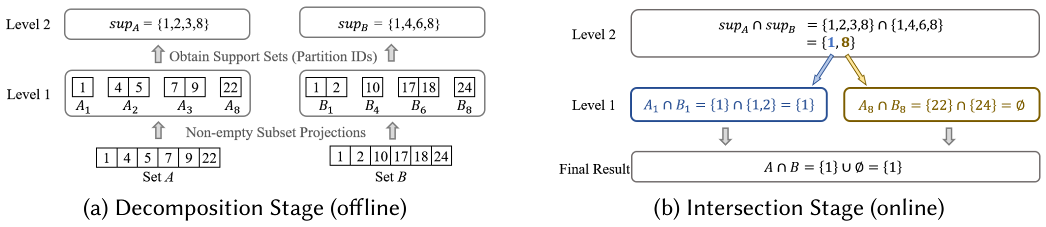

Level 2
$sup_A = \{1,2,3,8\}$
$sup_B = \{1,4,6,8\}$
Obtain Support Sets (Partition IDs)
Level 1
A_1 A_2 A_3 A_8
B_1 B_4 B_6 B_8
Non-empty Subset Projections
Set A
Set B
(a) Decomposition Stage (offline)
$sup_A \cap sup_B = \{1,2,3,8\} \cap \{1,4,6,8\} = \{1, 8\}$
$A_1 \cap B_1 = \{1\} \cap \{1,2\} = \{1\}$
$A_8 \cap B_8 = \{22\} \cap \{24\} = \emptyset$
Final Result
$A \cap B = \{1\} \cup \emptyset = \{1\}$
(b) Intersection Stage (online)

Fig. 2. An example of two-level set intersection.

subtasks such that each subtask can be efficiently solved by a single instruction. The SIB-tree can facilitate the multi-level set intersection efficiently.

In fact, many different SIB-trees can be built based on distinct set partitionings, and thus affect the intersection performance. We find that any particular set partitioning can be achieved by applying the graph reorder technique. Therefore, we formulate a novel graph reordering task and develop an efficient heuristic algorithm to optimize the performance of our proposed HERO approach, by minimizing the total size of the SIB-tree for all vertices in the graph.

In summary, we make the following contributions in this paper.

(1) **A novel set intersection framework**: We propose a novel *hierarchical set partitioning and join* (HERO) framework in Section 3 that decomposes the set intersection task into small-sized subtasks and filters out unnecessary comparisons.
(2) **A well-designed index for the set intersection**: We build an effective *set intersection bitmap tree* (SIB-tree) index for set intersection in the graph in Section 4, which can support efficient accomplishment of (sub)set intersection (sub)tasks via boolean AND operations over bitmaps.
(3) **Graph ordering optimization**: We formulate a novel graph reordering problem and propose a heuristic algorithm to optimize the performance of the proposed HERO approach in Section 5.
(4) **Remarkable empirical studies**: We have conducted extensive experiments in Section 6 to evaluate the proposed HERO approach. The results show that the speedup of our approach over the classic merge intersection achieves up to 188x and 176x for triangle counting and maximal clique enumeration, respectively.

2 PROBLEM DEFINITION

In this section, we formally define several concepts used for the set intersection problem.

Set Over the Graph. A graph is denoted as $G = (V, E)$, where V and E represent the sets of vertices and edges, respectively. We define a (vertex) set in the graph G below.

Definition 2.1 (Set in the Graph). Given a graph $G = (V, E)$, a set A in the graph refers to a subset of V, i.e., $A \subseteq V$.

One example of the set A in graph G can be a set of neighbors of a vertex $u \in V$. In this paper, we assume that each vertex u is assigned with a unique integer ID, and all vertices in a set A in graph G are sorted (e.g., in ascending order).

Set Intersection Problem. Next, we define the set intersection over the graph as follows.

Definition 2.2 (Set Intersection in the Graph). Given two sets A and B in the graph G, the set intersection task returns a set, $A \cap B$, of common vertices between two sets A and B, i.e., $A \cap B = \{u \in V \mid u \in A \wedge u \in B\}$.

As discussed above, two classical methods, merge intersection and galloping search, can be used to handle the set intersection problem in graphs. However, they often fail to fully leverage the intrinsic relationships among vertices, which can be improved. For instance, the merge intersection often involves unnecessary comparisons, leading to inefficiency. Furthermore, due to the limitation of comparing only one pair of elements per machine instruction, they are unable to fully exploit parallel advantages offered by bit-level computations. To address these limitations, we develop a novel HERO framework, which effectively filters out unnecessary comparisons and seamlessly integrates efficient boolean AND operations.

3 THE HIERARCHICAL SET PARTITIONING AND JOIN FRAMEWORK

In Section 3.1, we introduce a two-level intersection method that decomposes set intersection into two subtasks of intersection to be performed in two stages. We then expand the two-level method to a multi-level intersection framework by recursively abstracting set intersection to a higher-level task in Section 3.2.

3.1 Set Partitioning and Join

Set Partitioning: Let S be a universal set that contains all possible elements in the domain of set elements (e.g., all the vertex IDs in the graph). Given a positive integer n, our two-level set partitioning and join approach partitions the universal set S into n disjoint subsets $S_1, S_2, \dots,$ and S_n. We also call such subsets, S_i, *space partitions* of the set S, where i is the partition ID of the subset S_i.

Definition 3.1 (Subset Projection). Given a set $A \subseteq S$ and a subset S_i of S, the *subset projection*, A_i, of A over S_i contains common vertices between A and S_i, i.e., $A_i = A \cap S_i$.

Problem Reduction Based on the Set Partitioning: Given any two sets $A \subseteq S$ and $B \subseteq S$ to be intersected, it holds that:

$$\begin{aligned} A \cap B = (A \cap B \cap S) &= \bigcup_{i=1}^{n} \left(A \cap B \cap S_i\right) \\ &= \bigcup_{i=1}^{n} (A \cap S_i) \cap \bigcup_{i=1}^{n} (B \cap S_i) \\ &= \bigcup_{i=1}^{n} (A_i \cap B_i), \end{aligned} \tag{1}$$

where A_i and B_i (for $1 \leq i \leq n$) are the subsets projected from A and B onto the partition S_i, respectively, that is, $A_i = A \cap S_i$ and $B_i = B \cap S_i$.

Note that, when $A_i = \emptyset$ or $B_i = \emptyset$ holds, we have $A_i \cap B_i = \emptyset$. Therefore, in this case, we can actually avoid the set intersection cost of the i-th subtask $A_i \cap B_i$, if either A_i or B_i is empty.

Partition ID Join: Based on the observation above, we can record whether or not subsets A_i (or B_i) are empty, by keeping their partition IDs i in a so-called *support set* defined below (i.e., partition ID set) Sup_A (or Sup_B) if A_i (or B_i) are not empty. Then, we can join the two support sets Sup_A and Sup_B to obtain a list of subset pairs (or partition IDs) that need to perform the actual set intersection (i.e., subtasks).

Below we give the definition of the support set.

Definition 3.2 (Support Set). Given two sets A and B to be intersected, their support sets, Sup_A and Sup_B, over the space partition $S_1, S_2, \dots, S_n$ of the universal set S are defined as $\text{Sup}_A = \{i \mid A_i \neq$

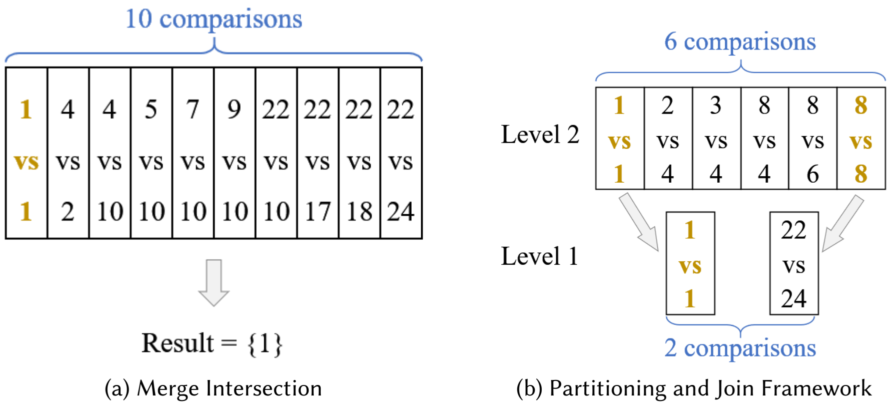


Fig. 3. Comparing costs of merge intersection with two-level set partitioning and join intersection

$\emptyset$, for $1 \leq i \leq n\}$ and $\text{Sup}_B = \{i \mid B_i \neq \emptyset, \text{for } 1 \leq i \leq n\}$, where A_i and B_i are subset projections for A and B over S_i, respectively.

Further Problem Reduction Based on the Partition ID Join: By using the concept of the support set (as given in Definition 3.2), we can rewrite the set intersection task in Equation (1) as:

$$A \cap B = \bigcup_{i \in (\text{Sup}_A \cap \text{Sup}_B)} (A_i \cap B_i). \tag{2}$$

Two-Level Intersection (Set Partitioning and Join): From Equation (2), we can: 1) first conduct the partition ID join, $\text{Sup}_A \cap \text{Sup}_B$, over the two support sets (Level 2, partition ID join), and 2) then perform the set intersection, $A_i \cap B_i$, only on those non-empty subsets A_i and B_i (Level 1, subset intersection), where partition IDs i are in the join result $\text{Sup}_A \cap \text{Sup}_B$. The above two steps exactly correspond to two levels of set intersection join, respectively.

In summary, *our principle is that the original task of the set intersection is decomposed into smaller subtasks of set intersections on two levels.* We will later discuss in Section 3.2 how to generalize this principle to that of multiple levels in a hierarchical structure.

Example 3.3. Assume that we have a universal set $S = \{v \in \mathbb{Z} \mid 1 \leq v \leq 27\}$, which can be partitioned into 9 disjoint subsets, $S_1 \sim S_9$, where $S_i = \{v \in S \mid 3(i-1) < v \leq 3i\}$ for $1 \leq i \leq 9$.

Given two sets $A = \{1, 4, 5, 7, 9, 22\}$ and $B = \{1, 2, 10, 17, 18, 24\}$, as shown in Figure 2(a), we first compute subset projections, A_i, of A over 9 partitions S_i. That is, we have $A_1 = \{1\}$, $A_2 = \{4, 5\}$, $A_3 = \{7, 9\}$, and $A_8 = \{22\}$ (note: other subsets $A_i = \emptyset$ for $i \neq 1, 2, 3, 8$). Similarly, for set B, we have non-empty subset projections $B_1 = \{1, 2\}$, $B_4 = \{10\}$, $B_6 = \{17, 18\}$, and $B_8 = \{24\}$.

Hence, the support set of A is $\text{Sup}_A = \{1, 2, 3, 8\}$, whereas that of B is $\text{Sup}_B = \{1, 4, 6, 8\}$.

In order to do the set intersection task $A \cap B$, as illustrated in Figure 2(b), we first compute the support set intersection $\text{Sup}_A \cap \text{Sup}_B = \{1, 8\}$ on the support set level (Level 2), and then perform the intersection of subsets for those partition IDs appearing in $\text{Sup}_A \cap \text{Sup}_B$ (i.e., 1-st and 8-th partitions), that is, $A \cap B = \bigcup_{i \in \{1,8\}} (A_i \cap B_i) = \big(A_1 \cap B_1\big) \cup \big(A_8 \cap B_8\big) = \{1\} \cup \emptyset = \{1\}$ (Level 1).■

Superiority of Two-Level Set Partitioning and Join: We illustrate the superiority of the two-level set intersection join mentioned above, by using Example 3.3. Figure 3 shows the computation costs of merge intersection [12] and our two-level set intersection frameworks, where each line segment represents one comparison between two elements in the sets. We can see that our framework

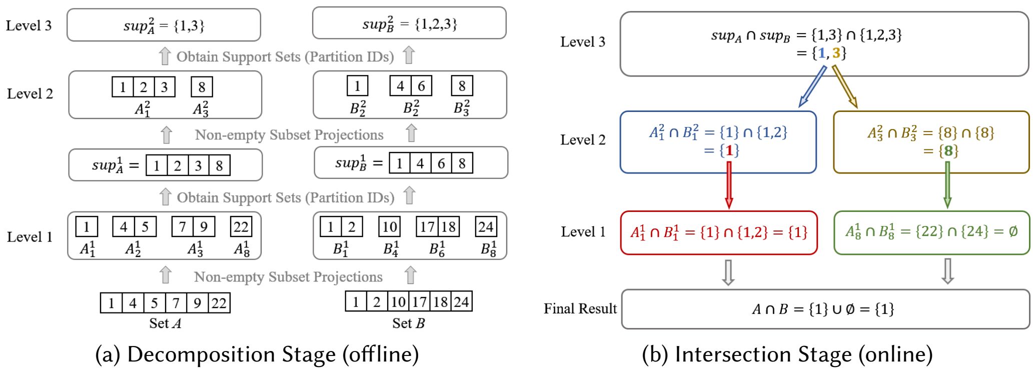

Level 3
$sup_A^2 = \{1,3\}$
$sup_B^2 = \{1,2,3\}$
Obtain Support Sets (Partition IDs)
Level 2
A_1^2 A_3^2
B_2^2 B_2^2 B_3^2
Non-empty Subset Projections
$sup_A^1 =$ 1 2 3 8
$sup_B^1 =$ 1 4 6 8
Obtain Support Sets (Partition IDs)
Level 1
A_1^1 A_2^1 A_3^1 A_8^1
B_1^1 B_4^1 B_6^1 B_8^1
Non-empty Subset Projections
Set A
Set B
(a) Decomposition Stage (offline)
$sup_A \cap sup_B = \{1,3\} \cap \{1,2,3\} = \{1,3\}$
$A_1^2 \cap B_1^2 = \{1\} \cap \{1,2\} = \{1\}$
$A_3^2 \cap B_3^2 = \{8\} \cap \{8\} = \{8\}$
$A_1^1 \cap B_1^1 = \{1\} \cap \{1,2\} = \{1\}$
$A_8^1 \cap B_8^1 = \{22\} \cap \{24\} = \emptyset$
Final Result
$A \cap B = \{1\} \cup \emptyset = \{1\}$
(b) Intersection Stage (online)

Fig. 4. An example of the Hierarchical Set Partitioning and Join (HERO) framework

only takes 8 (=6+2) comparisons, which is lower than 10 comparisons of the merge intersection algorithm. This is because the set intersection over some partitions (e.g., Partitions 2, 3, 4, and 6) can be avoided, and thus their computation costs can be saved.

From Figure 3, we can see that the major cost of our two-level set partitioning and join is now on Level 2 (i.e., 6 out of 8). It inspires us to further reduce the cost on this support set level, by introducing hierarchical set partitioning and join in the next section.

3.2 The Hierarchical Set Partitioning and Join (HERO) Framework

In this section, we generalize our idea of set intersection partitioning and join (as mentioned in Section 3.1) from two levels to multiple levels in a hierarchical structure. Algorithm 1 shows our hierarchical set partitioning and join (HERO) framework for the set intersection operator. It has two stages, i.e., offline set decomposition (lines 1-8) and online set intersection stages (lines 9-13).
Offline Set Decomposition: Given a set intersection task $A \cap B$, we consider an h-level intersection framework. Specifically, similar to the two-level case discussed in Section 3.1, on Level 1, we decompose the universal set S into n_1 disjoint partitions $S_1^1, S_2^1, \ldots,$ and $S_{n_1}^1$, and obtain projected subsets $\{A_i^1\}_{i=1}^{n_1}$ and $\{B_i^1\}_{i=1}^{n_1}$ from sets A and B, respectively. For ease of presentation, we use the superscript to denote the level number.

Then, on Level 2, we maintain two support sets, Sup_A^1 and Sup_B^1, containing partition IDs of non-empty projected subsets A_i^1 and B_i^1. The universal set on this level becomes $S^2 = \{j \in \mathbb{Z} | 1 \le j \le n_1\}$. In order to enable the intersection between support sets, $\text{Sup}_A^1 \cap \text{Sup}_B^1$, similar to Level 1, we recursively divide the new universal set into at most n_2 partitions $S_1^2, S_2^2, ...,$ and $S_{n_2}^2$, and project Sup_A^1 and Sup_B^1 onto these partitions, leading to A_i^2 and B_i^2, where $A_i^2 = \text{Sup}_A^1 \cap S_i^2$ and $B_i^2 = \text{Sup}_B^1 \cap S_i^2$ (for $1 \le i \le n_2$).

In general, on Level l ($2 < l \le h$), we maintain two support sets Sup_A^{l-1} and Sup_B^{l-1}, containing partition IDs of non-empty projected subsets A_i^{l-1} and B_i^{l-1}. The universal set in this level should be $S^l = \{j \in \mathbb{Z} | 1 \le j \le n_{l-1}\}$ and we divide it into n_l partitions $S_1^l, S_2^l, \ldots,$ and $S_{n_l}^l$. The support sets Sup_A^{l-1} and Sup_B^{l-1} will be projected onto these partitions, leading to the projected subsets A_i^l and B_i^l, where $A_i^l = \text{Sup}_A^{l-1} \cap S_i^l$ and $B_i^l = \text{Sup}_B^{l-1} \cap S_i^l$ (for $1 \le i \le n_l$).
Online Set Intersection: In contrast to the offline set decomposition, the online set intersection operates following a top-down paradigm. Initially, R^h is calculated directly by intersecting the two support sets, $\text{Sup}_A^h \cap \text{Sup}_B^h$, (line 8). The filtering result R^l determines which subsets in the l-th level subtasks need to be joined, where l ranges from h to 1. That is, for any element i in R^l, we

Algorithm 1: The Hierarchical Set Partitioning and Join (HERO) Framework

```
Input: a universal set $S$, set $A$, set $B$, and level $h$
Output: the intersection result $R = A \cap B$
// offline set decomposition
1  Initialize $S^1$ as $S$, $\text{Sup}_A^0$ as $A$, $\text{Sup}_B^0$ as $B$
2  for $l = 1$ to $h$ do
3      $S_1^l, \ldots, S_{n_l}^l \leftarrow$ partition the universal set $S^l$ into $n_l$ disjoint subsets
4      $A_1^l, \ldots, A_{n_l}^l \leftarrow$Project $(\text{Sup}_A^{l-1}, \{S_i^l\}_{i=1}^{n_l})$
5      $B_1^l, \ldots, B_{n_l}^l \leftarrow$Project $(\text{Sup}_B^{l-1}, \{S_i^l\}_{i=1}^{n_l})$
6      $S^{l+1} \leftarrow \{i \in \mathbb{Z} | 1 \leq i \leq n_l\}$
7      $\text{Sup}_A^l \leftarrow \{i \in S^{l+1} | A_i^l \neq \emptyset\}$, $\text{Sup}_B^l \leftarrow \{i \in S^{l+1} | B_i^l \neq \emptyset\}$
// online set intersection
8  $R^h \leftarrow \text{Sup}_A^h \cap \text{Sup}_B^h$
9  for $l = h$ to $1$ do
10     $R^{l-1} \leftarrow$Join $(R^l, \{A_i^l\}_{i=1}^{n_l}\}, \{B_i^l\}_{i=1}^{n_l})$
11 return $R^0$
// Details of the invoked procedure
12 Procedure Project$(X, \{Z_i\}_{i=1}^n)$
13     for $i = 1$ to $n$ do
14         $X_i \leftarrow Z_i \cap X$
15     return $\{X_i\}_{i=1}^n$
16 Procedure Join $(R, \{X_i\}_{i=1}^n, \{Y_i\}_{i=1}^n)$
17     $R' \leftarrow \emptyset$
18     for $i \in R$ do
19         $R' \leftarrow R' \cup$ Intersect$(X_i, Y_i)$
20     return $R'$.
```

need to solve the subtask Intersection(A_i^l, B_i^l) and add the results into R^{l-1}, while other subtasks can be safely skipped (lines 17-19). The above process also holds for $l = 1$, where the R^{l-1} (i.e., R^0) denotes result of $A \cap B$. The involved procedure Intersection(X, Y) is used to complete the subtask of computing the intersection between two sets X and Y. Notice that any particular algorithm for set intersection, such as the merge intersection, can be applied.

Example 3.4. Following the settings of S, A, B, and the space partition of S in Example 3.3, we can easily have the following results.

- $\text{Sup}_A^1 = \{1, 2, 3, 8\}$ and $\text{Sup}_B^1 = \{1, 4, 6, 8\}$;
- $A_1^1 = \{1\}, A_2^1 = \{4, 5\}, A_3^1 = \{7, 9\}, A_8^1 = \{22\}$, and $A_l^1 = \emptyset$ for any $l \notin \text{Sup}_A^1$.
- $B_1^1 = \{1, 2\}, B_4^1 = \{10\}, B_6^1 = \{17, 18\}, B_8^1 = \{24\}$, and $B_l^1 = \emptyset$ for any $l \notin \text{Sup}_B^1$.

To deal with the support set intersection of Sup_A^1 and Sup_B^1, we define the 2nd level space partition as $S_1^2 = \{1, 2, 3\}$, $S_2^2 = \{4, 5, 6\}$ and $S_3^2 = \{7, 8, 9\}$, where the new universal set is $\{i \in \mathbb{Z} | 1 \leq i \leq 9\}$. Then, as illustrated in Figure 4(a), we project the support set Sup_A^1 and Sup_B^1 into the partitions, and the non-empty projections should be $A_1^2 = \{1, 2, 3\}$, $A_3^2 = \{8\}$, B_1^2=$\{1\}$, $B_2^2 = \{4, 6\}$ and $B_3^2 = \{8\}$. Correspondingly, the support set should be $\text{Sup}_A^2 = \{1, 3\}$ and $\text{Sup}_B^2 = \{1, 2, 3\}$. With the above

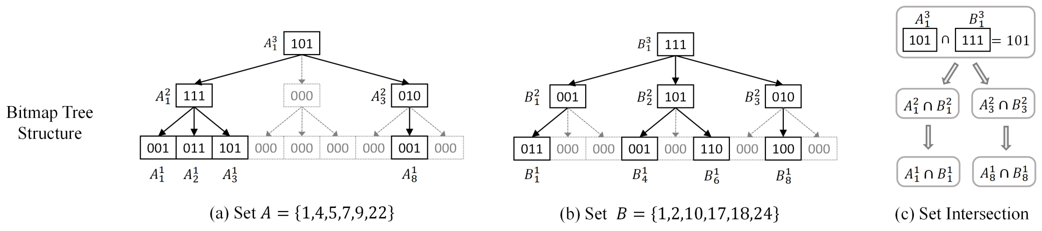


Fig. 5. An example of the SIB-tree index and intersection method. (a) The SIB-tree structure of set A; (b) The SIB-tree structure of set B; (c) The intersection procedure using the SIB-tree structure.

pre-computation, we can execute the multi-layer intersection as follows, which is also illustrated in Figure 4(b).

(1) $R^2 = \text{Sup}_A^2 \cap \text{Sup}_B^2 = \{1, 3\}$;
(2) $R^1 = \bigcup_{i\in R^2} \left(A_i^2 \cap B_i^2\right) = \left(A_1^2 \cap B_1^2\right) \cup \left(A_3^2 \cap B_3^2\right) = \{1, 8\}$;
(3) $A \cap B = \bigcup_{i\in R^1} \left(A_i^1 \cap B_i^1\right) = \left(A_1^1 \cap B_1^1\right) \cup \left(A_8^1 \cap B_8^1\right) = \{1\}$. ■

Time Complexity. The complexity of online set intersection in our HERO framework is highly related to a particular algorithm used in the subtask Intersection$(\cdot, \cdot)$. Assume that the complexity of procedure Intersection(A, B) is $O(f(|A|, |B|))$, where $|A|$ and $|B|$ are the size of sets A and B, respectively. The complexity of hierarchical online set intersection comprises of the cost of all the subtasks involved, i.e., $O(\sum_{l=1}^{h} \sum_{i\in R^h} f(|A_i^l|, |B_i^l|) + f(|\text{Sup}_A^h|, |\text{Sup}_B^h|))$.

4 BITMAP-BASED SET INTERSECTION

4.1 Set Intersection Using Boolean AND Over Bitmaps

In the HERO framework (as illustrated in Algorithm 1), one of the most fundamental and important operators is the subset intersections in subtasks (i.e., join two subsets to obtain their common elements). While we can plug in any existing set intersection method, in this paper, we use *bitmap* synopses to encode subsets and apply bit AND operator to enable the subset intersection.
Bitmap: Let d be the size of the set element domain. We define the bitmap, $BM(Z)$, for a subset Z as follows.

Definition 4.1 (Bitmap for a Subset, $BM(Z)$). Given a set Z, a *bitmap*, $BM(Z)$, of set Z is a bit vector containing d bits. If there exists an element $z \in Z$ such that $f(z) = pos$, then $BM(Z)[pos] = 1$ holds; otherwise, we have $BM(Z)[pos] = 0$, where $f(z)$ is a hash function that maps an element z to an integer pos within the range $[0, d)$.

Boolean AND Over Bitmaps: Given two subsets X_i and Y_i, we can obtain their intersection $X_i \cap Y_i$ (i.e., subtask) by using Boolean AND over their bitmaps, that is, $BM(X_i) \wedge BM(Y_i)$. Then, we can obtain those non-zero bit positions, pos, in the bitmap AND result, and their corresponding elements are common elements between subsets X_i and Y_i.
Discussions on the Advantages of Using Bitmaps for HERO:
Boolean AND operators are strictly constrained by the size of an operand register, also known as the word length. Thus, for the universal set S of large size (i.e., large element domain size), it is not efficient to directly perform Boolean AND operators over bitmaps of large size d. Assuming the word length is 64 and the graph contains more than a million vertices, it implies that more than 10,000 words are needed to encode a set using bitmaps, which leads to costly boolean AND operations to conduct a set intersection task. There are several methods that adopt the bitmap, such

as the compressed bitmap method Roaring [6] and the SIMD-based method QFilter [14]. However, these methods have to conduct merge operations that perform vertex-wise comparisons during the intersection procedure. In other words, the set intersection partially benefits from the advantage of bitmaps (i.e., supporting bit-wise parallel comparisons).

In contrast, our HERO framework recursively decomposes the set intersection task into small-sized subtasks, until each subtask can be efficiently handled by a single boolean AND instruction. Hence, our HERO is scalable for the intersection over sets of large domain sizes.

4.2 SIB-Tree Structure

In this subsection, we discuss how to design a *set intersection bitmap tree* (SIB-tree) structure to facilitate our proposed HERO framework for the set intersection.

Ordered Partitioning. Before introducing the details of the SIB-tree, we would like to unify the space partitioning in a natural way at first. To sufficiently leverage the advantage of the boolean AND operator, for subsets in varying-level space partitions, it is necessary to maintain a shared size of w which refers to the machine's word length. Given a universal set S^l, it can be simply partitioned into $n_l = \lceil \frac{n_{l-1}}{w} \rceil$ subsets $S^l_1, \ldots, S^l_{n_l}$, where $S^l_i = \{j \in S^l | w \cdot (i-1) < j \leq w \cdot i\}$ and $n_{l-1} = |S^l|$. We call this partitioning strategy an ordered partitioning. Clearly, the ordered partitioning is sensitive to the element orderings, and we will discuss the impact of various orderings in Section 5.

Let us recall the offline set decomposition in the HERO framework. Subset projections are performed from the l-th level to $(j+1)$-th level recursively. It naturally produces a tree structure, where each node represents a non-empty subset projection as shown in Figure 5. Following the tree, we can build an SIB-tree for each set to facilitate the boolean AND operation.

Definition 4.2 (Set Intersection Bitmap Tree, SIB-tree). Given a set A, its SIB-tree is denoted by $T(A)$, where each node $T^l_i(A)$ represents a non-empty subset projection A^l_i in a bitmap format and contains two attributes: *base value* and *bitmap value.*

- The base value, denoted by $T^l_i(A)$.base, is set to i, which implies that the elements of A^l_i fall in the range $(w \cdot (i-1), w \cdot i]$.
- The bitmap value, denoted by $T^l_i(A).bit$, is a w-bit bitmap whose k-th bit indicates whether or not the element $w \cdot (i-1) + k$ is involved in A^l_i.
- The root node of the SIB-tree $T(A)$ is $T^h_1(A)$, where $h = \lceil \log_w |S| \rceil$. The h-th level universal set S^h has no more than w elements, such that the corresponding h-th level subtask can be solved by a single boolean AND instruction.
- The nodes $\{T^1_i(A)\}^{n_1}_{i=1}$ constitute the leaf nodes.
- For each node $T^l_i(A)$, its child nodes are $\{T^{l-1}_u(A)\}_{u \in A^l_i}$.

It is clear that the l-th layer nodes of the SIB-tree $T(A)$ represent the l-th level non-empty subset projections $\{A^l_i\}_{i \in \mathrm{Sup}^l_A}$. Furthermore, the base value of a node tells which subset projection it represents in the corresponding level. Therefore, according to the HERO framework, the boolean AND operators only occur between nodes from two SIB-trees that locate on the same level and share the same base value.

Example 4.3. We follow the setting of S, A, B, and the space partition of S in Example 3.4. We first consider the bottom layer of $T(A)$. According to the Example 3.4, the non-empty 1st-level subset projections of A are $A^1_1 = \{1\}$, $A^1_2 = \{4, 5\}$, $A^1_3 = \{7, 9\}$, $A^1_8 = \{22\}$. Hence, there are 4 nodes $T^1_1(A)$, $T^1_2(A)$, $T^1_3(A)$, and $T^1_8(A)$ in the bottom layer of $T(A)$. The corresponding pairs of base value and bitmap value are $(1, 001)$, $(2, 011)$, $(3, 101)$, and $(8, 001)$, respectively. We then consider the 2nd layer. The non-empty 2nd-level subset projections of A are $A^2_1 = \{1, 2, 3\}$, and $A^2_3 = \{8\}$. Thus, the 2nd layer nodes are $T^2_1(A)$ and $T^2_3(A)$, together with the pairs of base value and bitmap value

$(1, 111)$, and $(3, 010)$, respectively. The 3rd level has only one subset projection $A_1^3 = \{1, 3\}$, which refers to the root node $T_1^3(A)$ with the base and bitmap value $(1, 101)$. We deliver an illustration of $T(A)$ in Figure 5(a). The construction of $T(B)$ is similar, so we skip the description and just show its illustration in Figure 5(b). ■

SIB-Tree Construction: Given a set A, we construct $T(A)$ in a bottom-up manner, progressing layer by layer. To construct the lowest layer (leaf nodes) of $T(A)$, it is necessary to compute subset projections $A_i^1 = A \cap S_i^1$, where i ranges from 1 to n_1. For each non-empty subset projection A_i^1, a leaf node $T_i^1(A)$ should be generated based on the SIB-tree definition. To establish the l-th layer of $T(A)$, where $2 \leq j \leq d$, we need to follow the subsequent steps.

(1) Collect the base value of the nodes in the $(j-1)$-th layer, where each node corresponds to a non-empty subset projection. Thus, all the base values in this layer constitute the support set Sup_A^{l-1}.
(2) Project Sup_A^{l-1} into the l-th level space partitions $\{S_i^l\}_{i=1}^{n_l}$, leading to the subset projections $\{A_i^l\}_{i=1}^{n_l}$.
(3) For each non-empty subset projection A_i^l, a new node $T_i^l(A)$ is created. For each $e \in A_i^l$, we set $T_i^l(A)$ as the parent node of $T_e^{l-1}(A)$.

Offline Complexity: For a wide range of graph analytics algorithms, the set intersection operations are conducted on the neighborhoods of several vertices. Therefore, we can construct an SIB-tree for the neighbors of each vertex during the offline stage. Since the total number of leaf nodes is no more than $|E|$, and the height of the SIB-tree is bounded by $O(\log |E|)$, the space complexity of such an offline computation is bounded by $O(|E| \log |E|)$. Furthermore, each SIB-tree node will be traversed at most once during the SIB-tree construction. Hence, the time complexity of the offline computation is also bounded by $O(|E| \log |E|)$.

4.3 SIB-Tree Based Intersection

Algorithm 2 shows how to calculate the set intersection using the SIB-tree structure in a depth-first manner. The algorithm employs a recursive approach, utilizing a helper function that takes a pair of vertices, denoted as (v_a, v_b), from $T(A)$ and $T(B)$ respectively, as input. These vertices share the same base value and depth (line 4).

Helper Function: The helper function first performs a boolean AND operation on the bitmap values of v_a and v_b (lines 5-6). Then, an iterator is created based on the shared base value of v_a and v_b, along with the result of the boolean AND operation (lines 15-20). The subsequent steps depend on whether the input vertices v_a and v_b are leaf vertices. If they are, each element in the iterator is directly added to the intersection result (lines 8-9). However, if they are not leaf vertices, the helper function iterates through each element in the iterator. If the iterator is empty, it backtracks to the last uncompleted subtask. For each element, it identifies the child vertices of v_a and v_b (referred to as child_a and child_b, respectively) whose base value matches the given element from the iterator (lines 12-13). Finally, the helper function recursively executes itself with child_a and child_b as input parameters (line 14).

Iterator Function: A base value and a bitmap value actually encode a set given a word length w. This iterator function is used to enumerate the elements contained in the set it encodes. The process is depicted in Algorithm 2 (lines 15-20). More specifically, in each iteration, the minimum element contained in the encoded set is computed by line 18, and then the bitmap value is updated by setting the first "1" bit to "0" (line 19).

Auxiliary Functions: Algorithm 2 includes two functions that require further clarification. The first function is $\text{FindChildNode}(\cdot)$. It takes a vertex v from the SIB-tree T and a query pos as inputs.

Algorithm 2: SIB-Tree Based Intersection

```
   Input: $T(A), T(B)$, word length $w$, and tree height $h$.
   Output: the result of intersection $R = A \cap B$.
1  res $\leftarrow \emptyset$
2  Helper $(T_1^h(A), T_1^h(B), \text{res})$
3  return res
4  Function Helper($v_a, v_b$, res):
5      $bit_a \leftarrow v_a.bit, \quad bit_b \leftarrow v_b.bit$
6      $bit \leftarrow \text{bit}_\text{a} \,\&\, \text{bit}_\text{b}$
7      base $\leftarrow v_a$.base
8      if $v_a$ and $v_b$ are leaf vertices then
9          res.extend(Iterator(base, $bit$, $w$))
10     else
11         for $i$ in Iterator (base, $bit$, $w$) do
12             child$_a$ $\leftarrow$FindChildNode $(v_a, i)$
13             child$_b$ $\leftarrow$FindChildNode $(v_b, i)$
14             Helper (child$_a$, child$_b$, res)
15 Function Iterator(base, $bit$, $w$):
16     res $\leftarrow \emptyset$
17     while $bit > 0$ do
18         res.push_back$\big((\text{base} - 1) * w + \texttt{Ffs}(bit)\big)$
19         $bit \leftarrow bit\&(bit - 1)$
20     return res
```

It returns a child vertex of v whose base value matches pos. The specific implementation details of FindChildNode$(\cdot)$ are dependent on the structure of the SIB-tree. The second function is Ffs$(\cdot)$, which is the acronym for "Find First Significant". Taking a bitmap as the input, the Ffs$(\cdot)$ function finds its lowest significant bit (i.e., the first "1" bit) and returns the corresponding index. With the help of Ffs$(\cdot)$ function, we can construct an iterator to enumerate the elements contained in the set encoded by a base value and bitmap value pair as discussed above. The GCC compiler provides efficient implementations of Ffs$(\cdot)$, such as the *__builtin_ffs*$(\cdot)$ for 32-bit unsigned integers and the *__builtin_ffsll*$(\cdot)$ for 64-bit unsigned integers.

Example 4.4. Based on the SIB-trees $T(A)$ and $T(B)$ constructed in Example 4.3, we illustrate how to conduct $A \cap B$ as shown in Figure 5(c). First, we input the pair of root nodes $(T_1^3(A), T_1^3(B))$ into the helper function. The boolean AND operator is conducted on their bitmap values, i.e., $101\&111 = 101$. The iterator function can help us decode the boolean AND result 101 and the base value 1 into the set $\{1, 3\}$. Hence, we need to find child node pairs $(T_1^2(A), T_1^2(B))$ and $(T_3^2(A), T_3^2(B))$, and feed them as input into the helper function sequentially. So far, we have accomplished a complete procedure of a helper function. The rest of the intersection process is similar. ■

Time Complexity: As outlined in Algorithm 2, the time cost of the SIB-tree based intersection mainly comes from the following three parts: (1) Computing intersection using boolean AND operator (lines 5-6). (2) Constructing the iterator (line 9 and line 11), whose time cost depends on the size of the iterator. (3) The invoking of FindChildNode$(\cdot)$ (lines 12-13) with the cost $O(1)$. It

is easy to find that each node in the SIB-tree will be traversed once at most in the intersection procedure. Furthermore, the times of invoking the boolean AND operator and FindChildNode$(\cdot)$ procedure are both bounded by the number of nodes that are traversed in the SIB-tree based intersection calculation. Hence, the time complexity is bounded by $O(|T(A)| + |T(B)|)$, where $|T(A)|$ represents the size of $T(A)$ (i.e., the number of nodes in $T(A)$). The construction of Iterator can be divided into leaf case (line 9) and non-leaf case (line 11). For the leaf case, the union of all the iterators equals R. Thus, the time complexity is bounded by $O(|R|)$, where R refers to the result of the intersection. As for the non-leaf case, it is clear that each element in the iterator corresponds to two FindChildNode$(\cdot)$ procedures. Therefore, the time cost is also bounded by $O(|T(A)| + |T(B)|)$. In summary, the time complexity of SIB-tree based intersection is $O(|T(A)| + |T(B)| + |R|)$.
Discussion on On-the-fly Intersection: If the sets for intersection are given on the fly, we have to construct the SIB-trees online. However, as long as the SIB-tree index is constructed, the subsequent intersection calculation can be accelerated. Especially, for the sets that are frequently involved in the intersection tasks, it will be beneficial if the corresponding SIB-trees are constructed.
Trade-off Between Online and Offline Computations. As mentioned in Section 4.2, we would like to construct SIB-trees for the neighbors of all vertices offline if it is allowed. However, if the offline computation resource (either time or space) is limited, there is an alternative to construct SIB-trees only for a subset of vertices. Intuitively, vertices that are frequently involved in the graph set intersection tasks are recommended to be selected.

5 GRAPH REORDERING

Section 4.2 recursively partitions/projects (support) sets onto subsets (partition ID subsets) by using the ordered partitioning as the space partition. Different space partitioning/projections may lead to different subsets, which in turn result in distinct SIB-tree performance. We find that different orderings of graph vertices may lead to different space partitions. Therefore, our basic idea of the *graph reordering* is to assign each vertex in the data graph G a new vertex ID, which equivalently reorders those vertices and obtains a better partitioning/projection strategy for the SIB-tree.

Example 5.1. We follow the settings of S, A, and B in Example 3.4. The reorder rule is illustrated at the top of Figure 6. Each vertex in S will be assigned a new ID, and the reordered set will be sorted according to the new IDs. Sets A and B are $\{1, 2, 3, 4, 5, 6\}$ and $\{1, 7, 8, 9, 10, 11\}$ after reordering, respectively. We also present the SIB-trees of reordered sets A and B in the Figure 6. The reordered SIB-trees of both sets A and B contain fewer tree nodes than the original SIB-trees in Figure 5 (4 v.s. 7, and 6 v.s. 8). Furthermore, the intersection task $A \cap B$ needs to handle only 3 subtasks based on new SIB-trees, which gains a significant improvement over the original order that has 5 subtasks.■

In fact, arbitrary partitioning can be exploited to build the SIB-tree only if it can produce equal-sized partitions $S_1, S_2, \ldots,$ and S_n. Any such a partitioning can obtained by performing the graph reordering and ordered partitioning. More specifically, we derive the vertex ID assignment rule $\phi : \mathbb{Z} \rightarrow \mathbb{Z}$, where for each vertex u's ID $u.ID \in S_i$ it holds that $\phi(u) = w \cdot (i - 1) + |\{v \in S_i | v.ID \leq u.ID\}|$. We can find that $\{\phi(v) | v \in S_i\} = \{j \in \mathbb{Z} | w \cdot (i - 1) < j \leq w \cdot i\}$. In other words, if we conduct the ordered partitioning according to new vertex IDs, the vertices contained in any S_i will still be contained in the same partition. Hence, the new partitioning results are equivalent to the original partitions.

In this section, we aim to develop a novel graph reordering method to optimize the performance of the proposed HERO approach. As discussed in Section 4.3, the time complexity of Algorithm 2 is bounded by the size of the SIB-tree as each vertex in the SIB-tree will be traversed at most once. Thus, ***the objective for the graph reordering task is to minimize the total size of the SIB-tree for all vertices*** v ***in the graph*** $G = (V, E)$**.** This target serves as a starting point in our approach

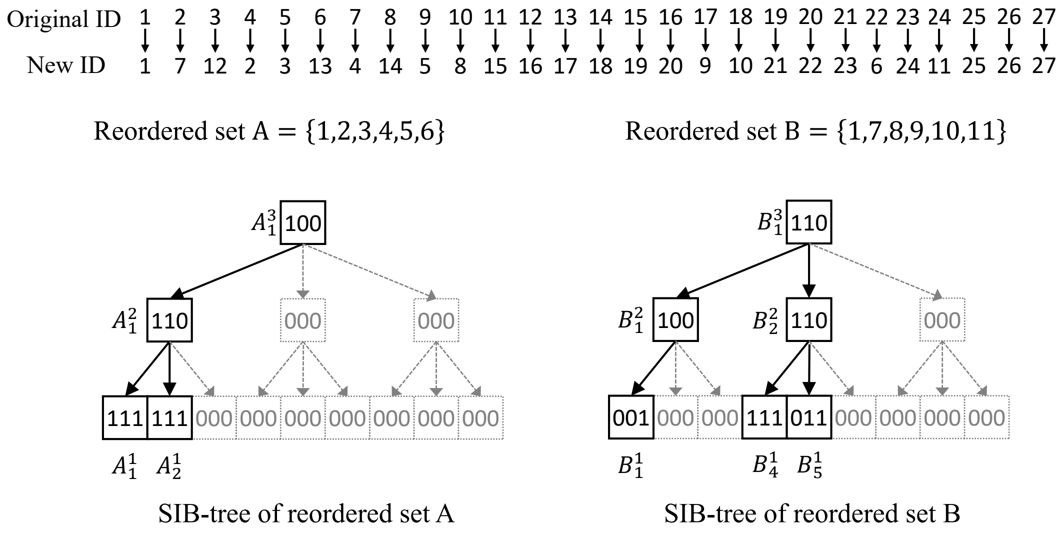

Original ID 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
New ID 1 7 12 2 3 13 4 14 5 8 15 16 17 18 19 20 9 10 21 22 23 6 24 11 25 26 27
Reordered set A = {1,2,3,4,5,6}
Reordered set B = {1,7,8,9,10,11}
A_1^3 100
A_1^2 110
000
000
111 111 000 000 000 000 000 000 000
A_1^1 A_2^1
SIB-tree of reordered set A
B_1^3 110
B_1^2 100
B_2^2 110
000
001 000 000 111 011 000 000 000 000
B_1^1 B_4^1 B_5^1
SIB-tree of reordered set B

Fig. 6. A motivating example of the reordering.

and can be mathematically formulated as follows:

$$\min \mathcal{L}(G) = \sum_{v \in V} |T(\mathcal{N}(v))|, \tag{3}$$

where $|T(\mathcal{N}(v))|$ is the size of the SIB-tree $T(\mathcal{N}(v))$. For simplicity, we would like to abbreviate $T(\mathcal{N}(v))$ as $T(v)$. Please note that the SIB-tree of the set which contains a single element v is denoted as $T(\{v\})$. Thus, the abbreviation will not lead to ambiguity.

Although we can consider different graph orders that lead to distinct SIB-trees and different intersection performance, it is computationally expensive to determine the optimal graph order (e.g., if we construct an SIB-tree for each graph order). To address this problem, we propose a *hierarchical balanced graph partitioning* (HBGP) strategy, which helps us compute the size of the SIB-tree index without constructing it. Next, we first introduce the definition of HBGP in Section 5.1. Then, we will describe how to compute the size of the SIB-tree index through HBGP, followed by the hardness analysis in Section 5.2. Finally, we develop an efficient heuristic algorithm to solve the HBGP problem in Section 5.3.

5.1 Hierarchical Balanced Graph Partitioning

Balanced Graph Partitioning: Given a graph $G = (V, E)$ and an integer n, the classical *balanced graph partitioning* (BGP) problem asks to split the vertices V into n disjoint and equal-sized subsets $P_1, \ldots, P_n$ such that the edge cut is minimized, where the edge cut is the number of the edges whose two vertices belong to different subsets [2]. The BGP is known as an NP-complete problem [2].

In BGP, the optimization target is to minimize the number of crossing edges, which can be reformulated as:

$$\frac{1}{2} \sum_{i=1}^{n} \sum_{u \in P_i} \left| \{v \in \mathcal{N}(u) | v \notin P_i\} \right| = \frac{1}{2} \sum_{i=1}^{n} \sum_{u \in P_i} \left| \mathcal{N}(u) \setminus P_i \right|. \tag{4}$$

By denoting the cost of a subset P_i as

$$\operatorname{cost}(P_i) = \frac{1}{2} \sum_{u \in P_i} \left| \mathcal{N}(u) \setminus P_i \right|, \tag{5}$$

Algorithm 3: General Greedy Paradigm for BGP

```
Input: Graph $G = (V, E)$, number of partitions $n$
Output: The partition result $\mathcal{P}$.
1  $w \leftarrow \lceil \frac{|V|}{n} \rceil$
2  Initialize $P_1, \ldots, P_n$ as $\emptyset$
3  $C \leftarrow V$
4  for $i = 1$ to $n$ do
5      while $|P_i| < w$ do
6          if $P_i = \emptyset$ then
7              $u \leftarrow$ Random pick from $C$
8          else
9              $u \leftarrow \arg\min_{u \in C} \text{cost}(P_i \cup \{u\})$
10         $C \leftarrow C \setminus \{u\}$
11         $P_i \leftarrow P_i \cup \{u\}$
12 return $\mathcal{P}$
```

we can rewrite the cost function of BGP as a general form

$$\mathcal{L}(\mathcal{P}) = \sum_{i=1}^{n} \text{cost}(P_i), \tag{6}$$

where $\mathcal{P} = \{P_1, \ldots, P_n\}$. Therefore, by defining different partitioning costs, it is easy to derive a wide class of generalized BGP. Note that we say the partitioning cost satisfies the *locality* if the cost of a partition is determined only by itself and is independent of other partitions, such that the cost of a partition can be computed during the construction procedure. Thus, the generalized BGP can be approximately solved by using a general greedy paradigm, as shown in Algorithm 3. It sequentially constructs the subsets following the greedy heuristic. Each subset P_i is initialized as an empty set (line 2), and a candidate set C is initialized as V (line 3). The construction of each subset P_i is an iterative procedure (lines 5- 11). In each iteration, a vertex in the candidate set C is selected (line 9) and inserted into the subset (line 11), such that the cost of the subset after insertion is maximized. Meanwhile, the vertex will be removed from the candidate set C once it has been selected (line 10).

Hierarchical Balanced Graph Partitioning: The graph partitioning problems above just divide the vertices through a unified granularity, however, it is required to partition the graph hierarchically in some cases. The hierarchy can be comprehended from two different perspectives. The first one is from top-down, that is, first dividing the graph into coarse-grained partitions and then dividing each partition into finer-grained partitions recursively. The second one is from bottom-up, that is, first dividing the graph into fine-grained partitions and then merging the partitions into coarser-grained partitions recursively. The partitioning result of HBGP is a balanced tree, where each node refers to a subset of the vertices, i.e., the union of the subsets referred to by its child nodes.

Definition 5.2 (Hierarchical Balanced graph partitioning problem, HBGP). Given a graph $G = (V, E)$ and a width parameter w, a h-level HBGP aims at partitioning V into h groups of subsets $\mathcal{P} = \{\{P_i^l\}_{i=1}^{n_l}\}_{l=1}^{h}$, such that

(1) $n_1 = \lceil \frac{|V|}{w} \rceil$. For $l = 2$ to h, $n_l = \lceil \frac{n_{l-1}}{w} \rceil$.

(2) For $l = 1$ to h, $\{P_i^l\}_{i=1}^{n_l}$ is a group of disjoint subsets of $|V|$, where each subset P_i^l has no more than $w \cdot j$ vertices.
(3) For $l = 2$ to h, each subset in $\{P_i^l\}_{i=1}^{n_l}$ is the union of w subsets in $\{P_i^{l-1}\}_{i=1}^{n_{l-1}}$ at most.
(4) The total cost $\mathcal{L}(\mathcal{P}) = \sum_{l=1}^{h} \sum_{i=1}^{n_l} \text{cost}(P_i^l)$ is minimized, where $\text{cost}(P_i^l)$ is defined as $|\bigcup_{u \in P_i^l} \mathcal{N}(u)|$.

Similar to the BGP, it is convenient to represent a result of HBGP by reordering the graph. That is, for each subset P_i^l, we always assign the ID of its vertices from $1 + w^{d+1-j} \cdot (i-1)$ to $w^{d+1-j} \cdot i$.

5.2 Computing SIB-Tree Size via HBGP

Given a hierarchical partitioning $\mathcal{P}$, we can build a bipartite graph $\mathcal{B} = (\mathcal{B}_l, \mathcal{B}_r, E_{\mathcal{B}})$ as follows. For each vertex u in G, it will be contained in the left part $\mathcal{B}_l$. While for each subset P_i^l in $\mathcal{P}$, we abstract it as a vertex in the right part $\mathcal{B}_r$. For each pair of vertices (u, P_i^l), we add an edge between them if and only if the intersection of the corresponding subset P_i^l and $\mathcal{N}(u)$ is non-empty. Let $T_i^l(u) \in T(u)$ denote that there exists a node in the l-th layer of SIB-tree $T(u)$ such that its base value equals i.

Lemma 5.3. *Given a graph $G = (V, E)$, a hierarchical partitioning $\mathcal{P}$ and the corresponding bipartite graph $A = (\mathcal{B}_l, \mathcal{B}_r, E_{\mathcal{B}})$, for each $u \in \mathcal{B}_l$ and $p_i^l \in \mathcal{B}_r$, it holds that $(u, p_i^l) \in E_{\mathcal{B}}$ if and only if $T_i^l(u) \in T(u)$.*

Proof. We just give a proof sketch of mathematical induction. For each $T_i^l(u) \in T(u)$, the target is to prove that $\mathcal{N}(u)$ contains at least one vertex in the P_i^l. It is trivial for $l = 1$. For the case $l \geq 2$, the key idea lies in that $T_i^l(u) \in T(u)$ is equivalent to $T_{i'}^{l-1} \in T(u)$ for at least one of the $w(i-1) < i' \leq wi$. Then, combining the proved case of $l-1$, the correctness of case l is achieved. □

Theorem 5.4. *Given a graph order, the optimization target $\mathcal{L}(G)$ defined in Equation (3) is equivalent to the cost of HBGP induced by the order. That is,*

$$\mathcal{L}(G) = \sum_{v \in V} |T(v)| = \mathcal{L}(\mathcal{P}) = \sum_{l=1}^{h} \sum_{i=1}^{n_l} \Big| \bigcup_{u \in P_i^l} \mathcal{N}(u) \Big|. \tag{7}$$

Proof. The proof can be achieved by counting the edges in the bipartite graph $\mathcal{B}$ in two different ways, i.e., summing up the degrees of all vertices in the left and right parts. Lemma 5.3 implies that the sum of degrees of the left part equals $\mathcal{L}(G)$. According to the definition of edges in $\mathcal{B}$, the sum of degrees of the right part equals $\mathcal{L}(\mathcal{P})$. Hence, the theorem is proved. □

Theorem 5.4 implies that optimizing the graph ordering is as difficult as solving the HBGP problem. which is NP-hard as shown next.

Theorem 5.5. *The hierarchical graph partitioning problem cannot be solved in polynomial time unless* P = NP.

Proof. The proof can be achieved by reducing from the *3-partition problem*, which is known as an NP-complete problem [13], to the HBGP problem. For each 3-partition problem instance $P1$, we can construct a corresponding HBGP instance $P2$. Then, determining whether P_1 has feasible solutions or not according to the solution of $P2$ will be given, and its correctness will be proved. In this way, we can prove that HBGP is not easier than the 3-partition problem, which implies that HBGP is NP. □

Due to space limitations, we will provide complete proofs of lemmas/theorems in a technical report after the paper publication (for the double-blind reason) and omit them here.

```
Algorithm 4: Greedy Algorithm for HBGP
    Input: Graph $G = (V, E)$, word length $w$
    Output: The hierarchical balanced graph partitions $\mathcal{P}$.
1   $h \leftarrow \lceil \frac{|V|}{w} \rceil, n_1 \leftarrow 1$
2   for $i = 2$ to $h$ do
3       $n_i \leftarrow \lceil \frac{|V|}{w^{d+1-i}} \rceil$
4       for $l = 1$ to $n_{i-1}$ do
5           left $\leftarrow w(j-1)+1$
6           right $\leftarrow \min(wj, n_i)$
7           Reorder (left, right, $w^{d+1-i}$)

8   Function Reorder(left, right, $w$):
9       $C \leftarrow \{v \in V | \text{left} \leq OID[v] < \text{right}\}$
10      $P \leftarrow \emptyset$
11      for $u$ = left to right do
12          if $P = \emptyset$ then
13              $v \leftarrow \arg\max_{v \in C} \deg(v)$
14          else
15              $v \leftarrow \arg\min_{v \in C} \text{cost}(P \cup \{v\})$
16          $OID[u] \leftarrow v$
17          $P \leftarrow P \cup \{v\}, C \leftarrow C \setminus \{v\}$
18          if $|P| = w$ then
19              $P \leftarrow \emptyset$
```

5.3 Heuristic Algorithm

Since the hierarchical balanced graph partitioning problem is intractable, we propose an efficient greedy heuristic algorithm. Based on the HBGP definition, a naive heuristic idea is to divide the HBGP problem into several BGP problems and solve them orderly, either top-down or bottom-up. In this paper, we adopt the top-down heuristic. The reason behind this is that the top-down heuristic prioritizes minimizing the number of high-level nodes in the SIB-tree. Consequently, space pruning is more likely to occur at the high-level intersection subtasks. The pseudocode of the heuristic algorithm is outlined in Algorithm 4, where the cost function in Line 15 is defined as

$$\text{cost}(P) = \left| \bigcup_{u \in P} \mathcal{N}(u) \right|. \tag{8}$$

For the procedure Reorder(left, right, w), we would like to store the set of candidate vertices C in a heap. Hence, each greedy selection (Line 13 and Line 15 in Algorithm 4) equals popping an element from the heap. The time complexity of a pop operation is $O(\log |C|)$. However, for each vertex v which is still in the heap, we need to update the $\text{cost}(P \cup \{v\})$ according to the Equation (8) such that we can correctly pop the next element. The fact is that the update procedure dominates the time complexity of Algorithm 4. Consequently, we need to design an efficient way for the update.

Consider line 15 in Algorithm 4, an equivalent way to make the greedy selection is computing the increment of the cost. That is, by denoting $\Delta(v|P) = \text{cost}(P \cup \{v\}) - \text{cost}(P)$, we can easily have

$$\arg\min_{v \in C} \text{cost}(P \cup \{v\}) = \arg\min_{v \in C} \Delta(v|P). \tag{9}$$

To demonstrate the advantage of using $\Delta(v|P)$ for update, we would like to rewrite it at first. Let $\mathcal{N}(P)$ denote $\bigcup_{p \in P} \mathcal{N}(p)$. We have

$$\begin{aligned} \Delta(v|P) &= \text{cost}(P \cup \{v\}) - \text{cost}(P) \\ &= \left|\mathcal{N}(P) \cup \mathcal{N}(v)\right| - \left|\mathcal{N}(P)\right| \\ &= \left|\mathcal{N}(v) \setminus \mathcal{N}(P)\right|. \end{aligned} \tag{10}$$

Then we compare the difference between $\Delta(v|P)$ and $\Delta(v|P \cup \{u\})$. According to the Equation (10), we have $\Delta(v|P) \geq \Delta(v|P \cup \{u\})$.

$$\begin{aligned} &\Delta(v|P) - \Delta(v|P \cup \{u\}) \\ =&\left|\mathcal{N}(v) \setminus \mathcal{N}(P)\right| - \left|\mathcal{N}(v) \setminus \left(\mathcal{N}(P) \cup \mathcal{N}(u)\right)\right| \\ =&\left|\mathcal{N}(v) \cap \left(\mathcal{N}(u) \setminus \mathcal{N}(P)\right)\right|. \end{aligned} \tag{11}$$

Equation (11) presents a direct and efficient update paradigm. It involves traversing the neighbors of each vertex in $\mathcal{N}(u) \setminus \mathcal{N}(P)$. Throughout this traversal, we keep track of the occurrence count for each vertex.

6 EXPERIMENTAL STUDY

6.1 Experimental Setting

Datasets. We use ten real graph datasets from SNAP [21] and KONECT [18] to evaluate the proposed SIB-tree based intersection method. Detailed statistics of graph datasets are listed in Table 1.

Tasks. We evaluate the SIB-tree based intersection method through the following three tasks.

(1) How does the SIB-tree based intersection accelerate the set intersection operation?
(2) How does the SIB-tree based intersection accelerate the graph algorithms that take the set intersection operation as a built-in block?
(3) How do different graph orders influence the performance of SIB-tree based intersection?

Competitors. We compare our proposed approach, denoted as SIB, with the following set intersection competitors over graphs.

(1) **Merge intersection** [12] is a classic method that scans to merge both sets at the same time;
(2) **Roaring bitmap** [6] is a compressed bitmap method, which is widely used in the database field;
(3) **QFilter** [14] is the state-of-the-art SIMD-based set intersection method, and;
(4) **Range** [31] is an efficient approach that adopts a reducing-merging framework.

Offline cost. Note that, all the competitors (except for the merge intersection) together with our SIB method are index-based approaches. We report the offline computation cost of these methods (i.e., the time of building the index for all vertices in a given graph) in Table 2. It shows that the SIB method is the most efficient for the index construction on most graphs (except for graphs TW, WG, CY, and SP). Furthermore, the space cost of these methods is also reported in Table 2. In the following subsections, we will focus on the online cost (instead of the offline cost).

Reproducibility. All experiments are conducted on a Linux Server equipped with Intel(R) Xeon(R) CPU E5-2640 @ 2.60GHz and 128G RAM. All algorithms used in the experiments are implemented

Table 1. Statistics about graphs used in the experiments, where $|V|$, $|E|$, and $\bar{d}$ denote the number of vertices, edges, and the average degree, respectively.

Graph	**Short Name**	**|V|**	**|E|**	**$\bar{d}$**
Twitter	TW	81,306	1,768,149	43.5
Gplus	GP	107,614	13,673,453	254.1
web-BerkStan	WB	685,230	7,600,595	22.2
web-Google	WG	875,713	5,105,039	11.7
com-Youtube	CY	1,134,890	2,987,624	5.3
soc-pokec	SP	1,632,803	30,622,564	37.5
as-skitter	AS	1,696,415	11,095,298	13.1
Flickr-links	FL	1,715,254	15,551,250	18.1
Wiki-Talk	WT	2,394,385	5,021,410	4.2
LiveJournal	LJ	3,997,962	34,681,189	17.3

Table 2. Offline performance of 4 methods for constructing indices on 10 graphs, including both time cost and space cost.

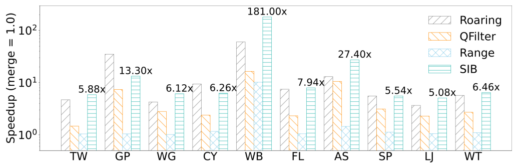
	Method	TW	GP	WG	CY	WB	FL	AS	SP	LJ	WT
Time (ms)	Roaring	325	628	1,663	1,032	746	2,116	2,606	5,687	13,305	1,460
	QFilter	**265**	2,354	**747**	**587**	1,173	2,996	2,128	**5,220**	8,890	834
	Range	42,524	1,734,520	269,423	392,762	3,174,943	2,478,530	3,216,431	3,645,284	12,308,925	5,487,410
	SIB	279	**379**	2,120	976	**259**	**1,389**	**1,070**	8,150	**4,716**	**634**
Space (MB)	Roaring	**6**	**48**	37	59	**28**	**103**	95	205	427	57
	QFilter	8	84	66	37	35	162	99	319	479	59
	Range	11	94	**27**	**36**	53	125	**91**	**176**	**342**	**45**
	SIB	12	67	105	85	54	277	221	622	1,135	162

in C++ and compiled by GCC v7.3.0 with -O3 option. Anonymous source code and data are available at: *https://anonymous.4open.science/r/HEROFramework-126E/*.

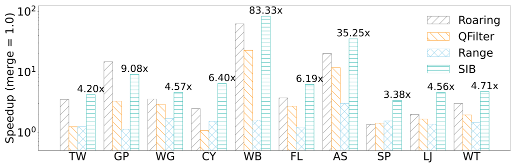
6.2 Accelerating Set Intersections

Following the previous research [31], we use two groups of query sets for global intersection and local intersection for each graph. The global intersection group contains 100, 00 pairs of randomly selected vertices to compute their common neighbors. The local intersection group contains 10, 000 pairs of randomly selected adjacent vertices to compute common neighbors. The results of global intersection and local intersection are presented in Figure 7 and Table 3.

Speedup of our approach. To evaluate the performance of the proposed SIB method, we compare it with the merge intersection and three state-of-the-art algorithms, that is, the compressed bitmap method Roaring [6], the SIMD-based method QFilter [14], and the reducing-merging method Range [31]. We report the speedups compared to the merge intersection as shown in Figure 7, where the elapsed time of the merge intersection is taken as the baseline with speedup=1.0x. We find that the SIB method consistently outperforms three baselines on 8 of 10 graphs for both global intersection and local intersection tasks. Specifically, on the graph web-BerkStan, the SIB method achieves significant speedups, up to 181.00x and 83.33x on global and local intersection tasks, respectively. Averaged on 10 graphs, the SIB method accelerates the global and local intersection tasks by 26.50x and 16.17x, respectively. We show the running time of the merge intersection algorithm in Table 4 (the columns "GI" and "LI" refer to global and local intersections, respectively). The running time of other algorithms can be easily inferred through the corresponding speedups.



(a) Global Intersection



(b) Local Intersection

Fig. 7. Speedup on global and local intersections.

Table 3. Number of comparisons ($\times 10^6$) involved in the merge intersection and our proposed SIB method, when performing the global intersection and local intersection tasks on 10 graphs.

Task	Method	TW	GP	WG	CY	WB	FL	AS	SP	LJ	WT
Global Intersection	Merge	18.7	121.3	47.8	47.8	170.0	123.7	163.7	19.9	79.9	141.9
	SIB	**0.9**	**8.0**	**5.7**	**2.1**	**0.2**	**8.2**	**1.6**	**1.5**	**7.3**	**9.5**
Local Intersection	Merge	10.6	79.6	27.4	22.3	155.7	79.8	134.3	15.1	47.5	13.4
	SIB	**0.4**	**2.6**	**0.3**	**0.3**	**0.2**	**2.4**	**0.2**	**0.4**	**1.2**	**0.4**

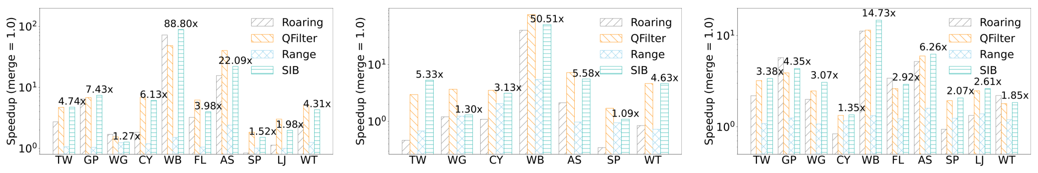
Table 4. The running time of each task on 10 graphs, using the merge intersection method. "–" indicates that the algorithm fails to complete within 10 days.

Graph	GI(ms)	LI(ms)	TC(ms)	MC(s)	SL(s)
TW	47	21	682	216	177
GP	665	236	58,289	–	190,856
WG	263	32	1312	19	143
CY	98	32	2,447	22	438
WB	181	250	28,506	842	4,274
FL	389	130	49,627	–	84,087
AS	274	141	39,583	1,699	5,699
SP	72	27	10,337	71	1,503
LJ	249	73	26,780	–	9,414
WT	439	33	4,257	1,856	2,560

The number of comparisons. Table 3 reports how many comparisons each intersection method demands on the global and local intersection tasks. For the merge intersection method, one comparison just refers to comparing a pair of vertices. For our proposed SIB method, one comparison refers to invoking the boolean AND operator once. Although the specific comparison operations are different, we find that there is a significant gap in the number of comparisons. For both global intersection and local intersection tasks, the merge intersection takes 10x-100x comparisons more than our proposed SIB, which may explain why the SIB method is much faster than the merge intersection method.

6.3 Accelerating Graph Algorithms

In this section, we evaluate the performance of SIB method through three graph algorithms, that is triangle counting (TC), maximal clique enumeration (MC), and subgraph listing (SL). For each task, we report the speedups compared with the merge intersection method, where the merge intersection is taken as the baseline with speedup= 1.0x.



(a) Speedups on TC task (b) Speedups on MC task (c) Speedups on SL task

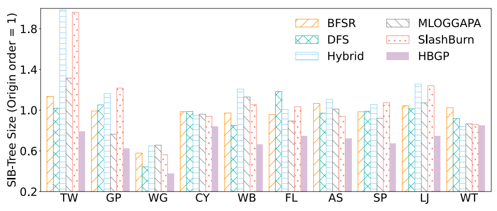
Fig. 8. Speedups over the merge method in downstream tasks TC, MC, and SL.

Table 5. Overall speedups over merge method on 10 graphs.

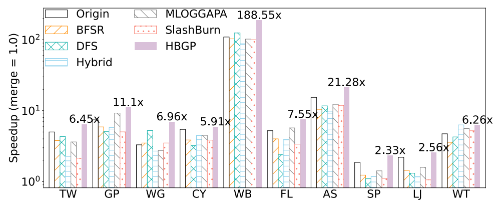
Task	Method	Average	Maximum	Median	Minimum
TC	Roaring	10.82	72.35	2.56	0.84
	QFilter	12.45	48.15	**5.65**	**1.54**
	Range	4.84	31.64	1.54	0.73
	SIB	**14.23**	**88.80**	4.53	1.27
MC	Roaring	6.59	40.08	1.09	0.35
	QFilter	**14.14**	**75.24**	3.68	**1.70**
	Range	2.55	7.28	1.56	0.86
	SIB	10.23	50.51	**4.63**	1.09
SL	Roaring	3.50	11.20	2.20	0.94
	QFilter	3.70	11.47	2.55	1.33
	Range	1.25	1.60	1.23	1.07
	SIB	**4.26**	**14.73**	**2.99**	**1.35**

Detailed speedups on each graph. Figure 8 illustrates the detailed performance of speedups on each graph. Our proposed SIB approach achieves remarkable speedups in most cases. In all 3 tasks, SIB consistently obtains higher speedups than Range on 10 graphs. Similarly, SIB outperforms Roaring in most cases (except for the SL task on GP, FL, and WT). Even if our SIB approach does not utilize any SIMD instruction, it is still competitive compared to the SIMD-based SOTA QFilter. Our SIB approach outperforms QFilter on 10 graphs in the SL task. In the TC task, it also outperforms QFilter on TW, GP, and WB. While in the MC task, QFilter exhibits marginal superiority on most graphs (except for TW).

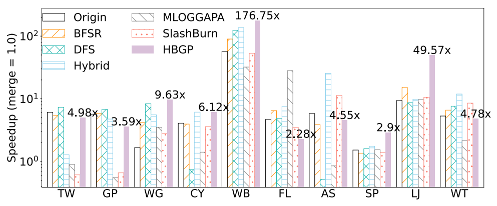
Overall speedups on each task. In addition, we present four overall statistics of speedups, namely the average, maximum, median, and minimum speedups, in Table 5. With respect to the average speedup, our SIB approach obtains 14.23x and 4.26x in TC and SL tasks, respectively, which outperforms all the competitors. In the MC task, its average speedup also reaches 10.23x, only less than QFilter (14.14x). As for the median speedup, which is more robust than the average, our SIB approach performs better than all the competitors in MC and SL tasks (4.63x and 2.99x), and achieves a second best in the TC task (4.53x). The maximum/minimum speedup is used to characterize the performance of each method in the best/worst cases. Our SIB approach obtains the best maximum speedups in TC and SL tasks (88.80x and 14.73x) and the best minimum speedup in the SL task (1.35x). As for the maximum (minimum) speedup in the MC (TC and MC) task, our SIB approach reaches the second best (50.51x, 1.27x and 1.09x), less than QFilter (75.24x, 1.54x and 1.70x).



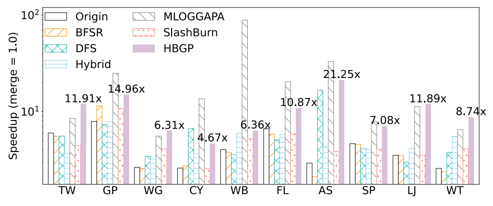
(a) Size reduction of SIB-tree index compared with the original order



(b) Speedups of SIB with different orderings on triangle counting task



(c) Speedups of SIB with different orderings on maximal clique task



(d) Speedups of SIB with different orderings on subgraph listing task

Fig. 9. Effect of graph orderings.

6.4 Evaluating Different Graph Orderings

In this section, we explore how different graph orderings benefit the SIB approach. In addition to the proposed ordering HBGP, we adopt 6 existing graph orderings, including the original order, BFSR [3], DFS [26], Hybrid [1], MLOGGAPA [11], and SlashBurn [22] as competitors, since they are often used for the graph set intersection in previous studies like QFilter [14] and Range [31]. We report the performance of the ordering through two metrics, that is, the size of the SIB-tree index and the speedup of SIB method on TC, MC, and SL algorithms. Note it is extremely expensive to enumerate MC on the whole graph, we enumerate MC on 1% vertices for each graph in this section.

Effect on the size of SIB-tree. The size information of the SIB-tree index is depicted in Figure 9(a). For simplicity of comparison, we scale the data by the size of the SIB-tree index in original order. We first consider the size of the SIB-tree index, which evaluates how well the ordering solves the graph reorder problem proposed in Section 5. As shown in Figure 9(a), our proposed HBGP-based ordering always constructs the SIB-tree index with minimal size among all the compared graph orderings. We also find that the performance of each order becomes pretty similar when the average degree of the graph is small. This phenomenon is intuitive because the small degree implies a corresponding small optimization space.

Effect on the speedups. As for the speedups of three graph algorithms, we report each ordering's average speedup over 10 graphs in Table 6. For detailed speedup on each graph, the results are shown in Figures 9(b), 9(c), and 9(d), respectively. Since the final target of reordering the graph is to improve the performance of SIB method, we then focus on the speedups brought by different orderings. According to the average speedups shown in Table 6, our proposed HBGP-based ordering beats all the ordering baselines in the TC and MC tasks. As for the SL task, our proposed HBGP ordering performs second only to the MLOGGAPA ordering. More specifically, the HBGP-based ordering performs the best on all 10 graph datasets in the TC task. Similarly, except for the MLOGGAPA

Table 6. Average speedups over merge method on 10 graphs.

Ordering	**TC**	**MC**	**SL**
Original	16.01	10.13	4.38
BFSR	14.09	14.26	4.45
DFS	16.34	17.03	5.92
Hybrid	12.54	21.09	4.82
MLOGGAPA	14.90	8.08	**22.03**
SlashBurn	13.73	9.63	4.86
HBGP	**25.90**	**26.52**	10.41

ordering, the HBGP-based ordering beats other ordering baselines on most of the graphs (8 of 10) in the SL task.

We notice that besides the intersection method, the ordering itself also influences the execution of all three tasks. To reduce this impact, we ensure that the times of intersection operations called in the TC or SL task using different graph orderings are consistent across each graph. However, this guarantee fails in the MC task, which may explain why the corresponding performance is irregular.

7 RELATED WORK

Set Intersection Acceleration: As we have mentioned before, existing researches about accelerating set intersection operation focus on how to leverage the SIMD hardware instructions for parallelism. Works such as SIMDShuffling and SIMDGalloping [19] directly equip the scalar. and galloping search algorithm with SIMD instructions, respectively. HieraInter [23] includes three algorithms for SIMD based merge intersection, which differ only in precision (8-bit, 16-bit and 32-bit). The BMiss [16] also improves the merge intersection method using SIMD instruction, meanwhile reduces the branch mispredictions. The graph processing engine EmptyHeaded [1] integrates both SIMDShuffling and SIMDGalloping algorithms and selects the better one at the run time. Roaring [6] is a widely used type of compressed bitmap index, and the corresponding library CRoaring [20] provides vectorized algorithms to compute union, intersection, difference, which supports the SIMD acceleration. The QFilter [14] is a two-level intersection method, which handles the high-level and low-level subtasks using SIMD based merge and bitmap intersection, respectively.

Graph Reordering: The graph reordering is a widely used technique for the graph compression, which reassigns vertex identifiers [3, 11, 13, 22, 26]. BFSR ordering [3] is a compact representation for separable graph, which reorders the vertices by building a separator tree recursively. DFS ordering [26] directly assigns vertices following the depth-first traversal. Hybrid ordering [1] first rearranges the graph by BFS. Then, vertices are sorted in descending order of the degree, where vertices with equal degree retain their BFS ordering. MLOGGAPA ordering [11] optimizes the logarithmic gaps of vertices in the same neighbor heuristically. Slashburn method [22] reorders the graph by iteratively removing hub vertices, such that its adjacency matrix consists of a few nonzero blocks.

8 CONCLUSION

In this paper, we focus on accelerating the set intersection operator on the graph data without the help of SIMD hardware instructions. We first propose the multi-level intersection framework, which recursively separates the set intersection task into small-sized subtasks and solves them independently. Then, we propose the SIB-tree index, which allow us to solve each subtask in the multi-level intersection framework using bitmap and boolean AND operation. Consequently, the intersection algorithm can totally get rid of the merge intersection framework.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (No. U23A20496 and No. 62072113), Shanghai Science and Technology Innovation Action Plan (No. 21511100401), and Natural Science Foundation (NSF CCF-2217104).

REFERENCES

[1] Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., and Ré, C. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst. 42*, 4 (2017), 20:1–20:44.

[2] Andreev, K., and Räcke, H. Balanced graph partitioning. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain* (2004), P. B. Gibbons and M. Adler, Eds., ACM, pp. 120–124.

[3] Blandford, D. K., Blelloch, G. E., and Kash, I. A. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA* (2003), ACM/SIAM, pp. 679–688.

[4] Brendel, W., Han, F., Marujo, L., Jie, L., and Korolova, A. Practical privacy-preserving friend recommendations on social networks. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018* (2018), P. Champin, F. Gandon, M. Lalmas, and P. G. Ipeirotis, Eds., ACM, pp. 111–112.

[5] Bron, C., and Kerbosch, J. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM 16*, 9 (1973), 575–576.

[6] Chambi, S., Lemire, D., Godin, R., and Kaser, O. Roaring bitmap : nouveau modèle de compression bitmap. In *Actes des 10e journées francophones sur les Entrepôts de Données et l'Analyse en Ligne, EDA 2014, Vichy, France, 5-6 Juin, 2014* (2014), S. Bimonte, L. d'Orazio, and E. Negre, Eds., vol. B-10 of *RNTI*, Hermann-Éditions, pp. 37–50.

[7] Chandran, J., and V., M. V. A novel triangle count-based influence maximization method on social networks. *Int. J. Knowl. Syst. Sci. 12*, 4 (2021), 92–108.

[8] Chu, S., and Cheng, J. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data 6*, 4 (2012), 17:1–17:32.

[9] Cui, W., Xiao, Y., Wang, H., Lu, Y., and Wang, W. Online search of overlapping communities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), K. A. Ross, D. Srivastava, and D. Papadias, Eds., ACM, pp. 277–288.

[10] Demaine, E. D., López-Ortiz, A., and Munro, J. I. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA* (2000), D. B. Shmoys, Ed., ACM/SIAM, pp. 743–752.

[11] Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., and Shalita, A. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016* (2016), B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds., ACM, pp. 1535–1544.

[12] Ding, B., and König, A. C. Fast set intersection in memory. *Proc. VLDB Endow. 4*, 4 (2011), 255–266.

[13] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[14] Han, S., Zou, L., and Yu, J. X. Speeding up set intersections in graph algorithms using SIMD instructions. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 1587–1602.

[15] Huang, M., Jiang, Q., Qu, Q., Chen, L., and Chen, H. Information fusion oriented heterogeneous social network for friend recommendation via community detection. *Appl. Soft Comput. 114* (2022), 108103.

[16] Inoue, H., Ohara, M., and Taura, K. Faster set intersection with SIMD instructions by reducing branch mispredictions. *Proc. VLDB Endow. 8*, 3 (2014), 293–304.

[17] Kang, J., Zhang, J., Song, W., and Yang, X. Friend relationships recommendation algorithm in online education platform. In *Web Information Systems and Applications - 18th International Conference, WISA 2021, Kaifeng, China, September 24-26, 2021, Proceedings* (2021), C. Xing, X. Fu, Y. Zhang, G. Zhang, and C. Borjigin, Eds., vol. 12999 of *Lecture Notes in Computer Science*, Springer, pp. 592–604.

[18] Kunegis, J. KONECT: the koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume* (2013), L. Carr, A. H. F. Laender, B. F. Lóscio, I. King, M. Fontoura, D. Vrandecic, L. Aroyo, J. P. M. de Oliveira, F. Lima, and E. Wilde, Eds., International World Wide Web

Conferences Steering Committee / ACM, pp. 1343–1350.

[19] Lemire, D., Boytsov, L., and Kurz, N. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exp. 46*, 6 (2016), 723–749.

[20] Lemire, D., Kaser, O., Kurz, N., Deri, L., O'Hara, C., Saint-Jacques, F., and Kai, G. S. Y. Roaring bitmaps: Implementation of an optimized software library. *Softw. Pract. Exp. 48*, 4 (2018), 867–895.

[21] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] Lim, Y., Kang, U., and Faloutsos, C. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Trans. Knowl. Data Eng. 26*, 12 (2014), 3077–3089.

[23] Schlegel, B., Willhalm, T., and Lehner, W. Fast sorted-set intersection using SIMD instructions. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2011, Seattle, WA, USA, September 2, 2011* (2011), R. Bordawekar and C. A. Lang, Eds., pp. 1–8.

[24] Shao, Y., Cui, B., Chen, L., Ma, L., Yao, J., and Xu, N. Parallel subgraph listing in a large-scale graph. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014* (2014), C. E. Dyreson, F. Li, and M. T. Özsu, Eds., ACM, pp. 625–636.

[25] Shoaran, M., and Thomo, A. Zero-knowledge-private counting of group triangles in social networks. *Comput. J. 60*, 1 (2017), 126–134.

[26] Shun, J. *Shared-memory parallelism can be simple, fast, and scalable.* Morgan & Claypool, 2017.

[27] Shun, J., and Tangwongsan, K. Multicore triangle computations without tuning. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015* (2015), J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds., IEEE Computer Society, pp. 149–160.

[28] Wang, N., Zhang, J., Tan, K., and Tung, A. K. H. On triangulation-based dense neighborhood graphs discovery. *Proc. VLDB Endow. 4*, 2 (2010), 58–68.

[29] Yaozu, Cui, Junqiu, Li, Xingyuan, and Wang. Uncovering the overlapping community structure of complex networks by maximal cliques. *Physica, A. Statistical mechanics and its applications 415* (2014), 398–406.

[30] Zechner, N., and Lingas, A. Efficient algorithms for subgraph listing. *Algorithms 7*, 2 (2014), 243–252.

[31] Zheng, W., Yang, Y., and Piao, C. Accelerating set intersections over graphs by reducing-merging. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021* (2021), F. Zhu, B. C. Ooi, and C. Miao, Eds., ACM, pp. 2349–2359.

Received July 2023; revised October 2023; accepted November 2023