GraphCPP: A Data-Driven Graph Processing System for Concurrent Point-to-Point Queries

IEEE Publication Technology Department

***Abstract*—With the widespread adoption of graph processing technology in fields such as map navigation and network analysis, a considerable number of point-to-point query tasks run concurrently on the same underlying graph, imposing a high demand on the throughput of graph query systems. However, existing graph processing systems either focus on optimizing the speed of individual point-to-point queries, neglecting the throughput of concurrent queries, or follow the concurrent execution schema of point-to-all algorithms, overlooking the optimization potential of point-to-point algorithms. Due to redundant data access overhead and computational expenses, existing solutions exhibit poor overall throughput when executing concurrent point-to-point queries.**

This paper introduces GraphCPP, the first graph processing system designed for concurrent execution of point-to-point query tasks. It enhances the throughput of concurrent query tasks through data access sharing and hot path computation sharing. GraphCPP introduces two novel features. Firstly, it incorporates a data-driven caching execution mechanism that leverages the significant overlap in traversal paths for various query tasks. This facilitates data sharing among concurrent tasks, ultimately enhancing data access efficiency. Secondly, GraphCPP addresses the recurrent computation of identical hot vertices and paths by distinct tasks through a dual-level computation sharing mechanism. This accelerates the convergence of new queries by efficiently sharing computed values of hot vertices and paths. In our evaluation on datasets such as xx, we compare GraphCPP with state-of-the-art point-to-point query systems and concurrent graph processing systems. The results demonstrate that GraphCPP incurs only xx preprocessing overhead and xx storage overhead, achieving a throughput improvement of xx-xx times compared to SGraph, Tripoline, Pnp, and Glign.

***Index Terms*—graph process, point-to-point queries, concurrent queries, data access similarity, computational similarity.**

I. Introduction

GENERALLY, large-scale concurrent point-to-point query tasks involve analyzing associated vertices between two specific locations on the same underlying graph. Notable instances encompass applications such as Google Maps[6], optimizing logistics routes. Facebook[7], suggesting potential friends through exploring relationship chains in social network analysis. and Alipay[8], analyzing risk propagation between entities in financial risk assessment. These applications underscore the imperative for efficiently executing concurrent point-to-point queries on a shared underlying graph. In contrast to conventional graph query methods, point-to-point queries concentrate on specific vertices, bypassing the complexity associated with querying the entire graph or its extensive subsets. This targeted approach sets point-to-point query algorithms apart from point-to-all query algorithms, presenting substantial optimization potential. However, prevalent general-purpose concurrent graph computing systems[4] typically employ traditional methods for point-to-point query execution, overlooking optimization opportunities for traversal processes. Existing specialized solutions for point-to-point queries[1][2][3] predominantly emphasize enhancing the efficiency of individual queries, often neglecting concurrent query optimization. To achieve efficient processing of concurrent point-to-point queries, two challenges must be addressed.

Firstly, when executing concurrent point-to-point query tasks, failure to consider data access similarity results in redundant access to overlapping data. Specifically, different query tasks initiate from distinct starting points and traverse different paths to reach their respective endpoints. Due to the power-law distribution characteristics of graphs, these query paths naturally overlap on some popular routes (More details are in Section x). However, since the overlapping portions of data accessed by different query tasks vary, they traverse along different paths to access overlapping graph data. Consequently, existing query systems adopt a conservative strategy, assigning each task the responsibility of accessing its required data. This implies that data access for each task is entirely independent, even if their traversal paths highly overlap. For example, if the path of one query pair is a subset of the path of another query pair, it still necessitates reloading data from the overlapping portions, preventing the benefits of reusing cached data.

Secondly, in addition to redundant access, different query tasks also need to redundantly compute distance values for popular paths. Similarly, due to the power-law distribution characteristics of graph data, a few popular vertices frequently appear in optimal paths for different queries (More details are in Section x). As different query tasks do not share computations, they redundantly calculate the hot paths connecting popular vertices. Moreover, popular vertices often possess a large number of neighboring vertices, and the repetitive traversal of these vertices leads to an explosive growth in computational expenses. Although some existing systems attempt to implement computation sharing through a global index[1][2], the efficiency and accuracy of computation sharing are restricted due to the expensive computational, storage, and update costs.

To address the challenges mentioned above, this paper proposes a data-driven concurrent point-to-point query system called GraphCPP. It enhances the throughput of concurrent point-to-point query systems through data sharing and computation sharing mechanisms.

In GraphCPP, we firstly introduce a data-driven caching execution mechanism that transforms the traditional $task \rightarrow$

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

$data$ scheduling approach into a $data \rightarrow task$ scheduling approach, thereby enabling the sharing of overlapping graph structure data among multiple tasks. Under this execution mechanism, GraphCPP first determines the scheduling order of data: it logically partitions graph structure data into fine-grained chunks at the LLC level. Subsequently, it associates query tasks with the relevant graph chunks based on the graph block where the active vertices of the query tasks reside. The higher the number of associated tasks, the higher the priority for scheduling that chunk. As the set of active vertices changes in each round, and the number of associated tasks for shared chunks needs updating in each round, GraphCPP adopts an associated task-triggering mechanism to achieve $data \rightarrow task$ scheduling. After loading the graph chunks into the LLC in priority order, the system utilizes the associated information obtained in each round to trigger the batch execution of associated tasks for the current chunk, efficiently accessing shared data.

GraphCPP proposes a dual-level computation sharing mechanism that incorporates a global index and a core subgraph index. It identifies vertices in the graph with a significant number of connecting edges as hot vertices and the paths between these hot vertices as hot paths. Despite the limited quantity of hot vertices and hot paths, they frequently appear in the traversal paths of different query tasks. 1) Global Index Mechanism: The global index maintains information about a small number of hot vertices and their connections to other vertices. These hot vertices serve as intermediary nodes for various query paths, providing pruning values for the majority of queries. Through the global index, the first level of computation sharing is achieved. 2) Core Subgraph Mechanism: The core subgraph primarily maintains hot paths connecting hot vertices in the graph. Notably, the number of hot vertices in the core subgraph is an order of magnitude larger than that in the global index, enabling a broader coverage. Specifically, it starts by filtering highly connected hot vertices from the original graph data. Subsequently, it extracts hot paths from the convergence results of each query, storing their values in the core subgraph. During the query process, the core subgraph functions like a highway network, facilitating traversal from one hot vertex to another. This process is akin to entering a highway from a hot vertex and utilizing the fast lanes (hot paths) on the highway network to swiftly reach other high-speed stations (hot vertices). Despite maintaining more hot vertices, the core subgraph's overall scale is significantly smaller than that of the entire graph. Additionally, the values in the core subgraph, representing hot paths between hot vertices, can be obtained without computation, resulting in minimal additional overhead during the query process.

Lastly, GraphCPP further enhances the performance of concurrent queries by predicting the traversal paths of different query tasks and driving the batch execution of highly overlapping similar query tasks.

This paper makes the following contributions:

1) We analyze the performance bottlenecks of existing graph processing systems when handling concurrent point-to-point query tasks and propose leveraging the similarities in data access and computation among concurrent tasks to enhance throughput.
2) We develop GraphCPP, a dynamic data-driven concurrent processing system for point-to-point queries. It achieves data sharing and computation sharing among concurrent tasks and introduces a strategy for batch execution of similar tasks.
3) we compare GraphCPP with state-of-the-art point-to-point query systems and concurrent graph processing systems, using a workload that includes six real-world graphs[14][15][16][17] and six applications[18][21][22][23][24][27]. Our experiments demonstrate that, GraphCPP improves their performance by 1.73 13 times.

System	Pnp				Tripoline				SGraph				Glign			
Number of concurrent query tasks	1	4	8	16	1	4	8	16	1	4	8	16	1	4	8	16
Instructions ($\times 10^{14}$)	7.28	7.41	7.32	7.4	5.9	5.68	6.1	5.57	3.69	3.54	3.28	3.73	1.55	1.76	1.49	1.61
LLC loads ($\times 10^{12}$)	28.3	27.6	28.5	27.9	15.7	16.2	15.6	15.9	8.2	8.6	8.5	8.4	5.7	5.5	5.8	5.4
LLC miss ratio	45.8%	52.6%	60.3%	70.2%	40.3%	52.4%	64.8%	72.4%	42.5%	57.1%	63.8%	69.5%	32.4%	47.2%	55.4%	67.3%
Runtime (hour)	72.96	30.29	27.65	20.11	53.42	19.76	12.31	8.84	27.24	11.96	7.28	5.36	15.47	6.18	4.32	2.19

TABLE I: Performance comparison of existing systems for processing concurrent point-to-point queries

II. BACKGROUND AND MOTIVATION

Most existing solutions[3][2][1] are primarily focused on accelerating the speed of individual queries. However, in practical scenarios, there is a significant number of graph query tasks concurrently running on the same underlying graph. For instance, statistics from CAGIS[9] indicate that location service open platforms constructed by companies such as Baidu Maps[10], Amap[11], Tencent Maps[12], and Huawei Maps[13] receive a daily average of up to 160 billion location service requests. The substantial demand for concurrent point-to-point queries poses a high requirement for the throughput of graph traversal systems. However, as illustrated in table 1, we demonstrate that existing systems exhibit low throughput when handling large-scale concurrent queries(Details are given in Section x). The root cause of this undesirable outcome is the significant redundancy in data access and computation among concurrent tasks. To qualitatively analyze the aforementioned issues, we conducted performance evaluations on parallel point-to-point queries using XXXXX (machine configuration), selecting XXXXX (the current best solution) on XXXXX (graph dataset).

A. Preliminaries

Definition 1: ($Graph$) We represent a directed graph using $G = (V, E)$ (edges in an undirected graph are considered as bidirectional edges), where V is the set of vertices, and E is the set of directed edges formed by vertices in V. We use $|V|$ and $|E|$ to denote the number of vertices and edges, respectively. In distributed environments, the graph partitioning problem also exists. We use $P_i = (V_{P_i}, E_{P_i})$ to represent the i_{th} partition of the directed graph, where V_{P_i}

denotes the set of vertices in the partition, and E_{P_i} is the set of directed edges formed by vertices in V_{P_i}.

Definition 2: ($Point\text{-}to\text{-}Point\ Query$) We use $q_i = (s_i, d_i)$ to denote the query corresponding to task i, where s_i and d_i represent the source and destination vertices of q_i, respectively. If the execution of the point-to-point query algorithm on the graph yields a convergent path for q_i, this path is referred to as the optimal path between s_i and d_i. The distinction between point-to-point queries and regular queries lies in the former's ability to reduce unnecessary traversals through pruning. Pruning involves using the value of the path between two points as a threshold, referred to as the bound. Iteratively traversing new paths updates the bound, and when the bound converges, it signifies the discovery of an optimal path.

Definition 3: ($Core\ Subgraph$) We denote the core subgraph as $G_{core} = (V_{hot}, E_{hot}, Index_{hot})$, where V_{hot} represents the set of hot vertices, E_{hot} represents the set of edges abstracted from hot paths, and $Index_{hot}$ represents the set of values associated with the hot paths. In this context, hot vertices refer to high-degree vertices, and hot paths are optimal paths connecting pairs of hot vertices.

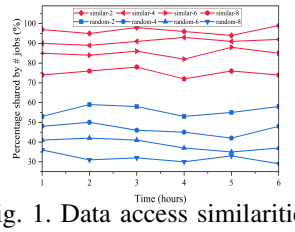

Fig. 1. Data access similarities between concurrent tasks

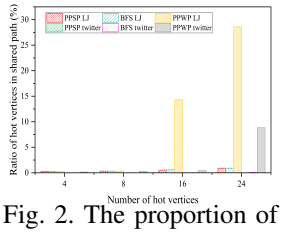

Fig. 2. The proportion of high-degree vertices in the overlapping data

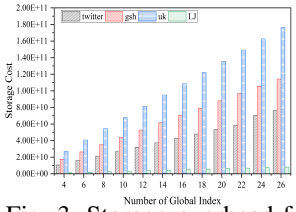

Fig. 3. Storage overhead for global indexes

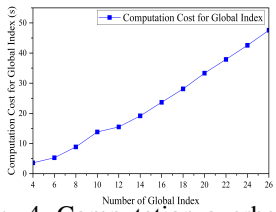

Fig. 4. Computation overhead for global indexes

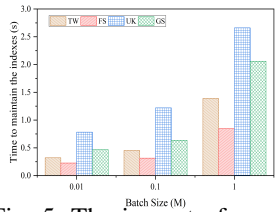

Fig. 5. The impact of graph updates on global indexes

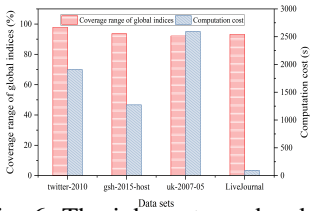

Fig. 6. The inherent overhead and effectiveness of a global index varies with k

B. Performance Bottlenecks in Concurrent Point-to-Point Query Tasks

In this section, we have implemented the concurrent version of the existing linear pairwise query system[3][2][1] and the pairwise version of the existing concurrent graph computing system[4]. Our objective is to quantitatively elucidate the performance bottlenecks of current concurrent pairwise query solutions.

Dataset	twitter-2010				gsh-2015-host				uk-2007-05				LiveJournal			
Number of global index	4	8	12	16	4	8	12	16	4	8	12	16	4	8	12	16
Storage Cost	7.28	7.41	7.32	7.4	5.9	5.68	6.1	5.57	3.69	3.54	3.28	3.73	1.55	1.76	1.49	1.61
Computation Cost	28.3	27.6	28.5	27.9	15.7	16.2	15.6	15.9	8.2	8.6	8.5	8.4	5.7	5.5	5.8	5.4
Coverage Range	45.8%	52.6%	60.3%	70.2%	40.3%	52.4%	64.8%	72.4%	42.5%	57.1%	63.8%	69.5%	32.4%	47.2%	55.4%	67.3%

TABLE II: Performance comparison of existing systems for processing concurrent point-to-point queries

Redundant data access overhead We observe that when concurrent pairwise query tasks execute graph traversal on the same underlying graph, a significant portion of their traversal paths exhibits clear data access similarities. As shown in Figure 3x, our data indicates that there is a substantial overlap in data access among concurrent tasks, with a higher overlap ratio for similar tasks (more details are provided in Section xx). However, in the traditional $task \rightarrow data$ scheduling mode, different tasks independently execute queries, leading to competition for limited cache space to store their respective graph data chunks, even if these data chunks are identical. This results in severe cache thrashing and a reduction in the performance of processing concurrent pairwise queries. Table 1 illustrates that, with an increase in the number of concurrent tasks, the cache miss rate for query tasks sharply rises. Although the overall performance of concurrent execution surpasses linear execution, the average query time per task significantly increases with the growing number of query tasks due to the mentioned redundant data access overhead. It is essential to note that the total execution time in the concurrent mode is the maximum among the execution times of these tasks, while in the sequential mode, it is the sum of all task execution times.

Redundant Computational Overhead Owing to the power-law distribution inherent in graph data, the traversal paths of query tasks involve a significant number of hot vertices. As depicted in Figure 2, despite hot vertices constituting only a small fraction of the total vertices (approximately XX%), they frequently appear in the traversal paths of multiple tasks (about XX%). Furthermore, the degree of sharing among hot vertices in graph data is positively correlated with their proportion in paths. Consequently, different query tasks may redundantly calculate paths between hot vertices, highlighting computational similarities between concurrent tasks. During a graph snapshot period, the calculation results for paths between identical vertex pairs remain consistent, indicating significant redundancy in computations among concurrent tasks. Moreover, due to the typically higher number of outbound and inbound edges associated with hot vertices, their computational overhead is considerably larger compared to regular vertices.

Certain existing solutions attempt to leverage computational similarity through a global indexing mechanism[2][1]. However, this mechanism inherently suffers from three key drawbacks: 1) The computational and storage costs of estab-

lishing a global index, recording path values between highly connected vertices and all others, escalate sharply with the graph's increasing scale, as depicted in Figures 3 and 4. This escalation is directly proportional to the number of vertices in the global index. 2) In dynamic graph processing, incremental updates refresh the global index following each round of graph updates. However, if affected vertices lie on an optimal path, the global index values for all vertices along the path from the directly impacted vertex to the endpoint must be recalculated. The impact of graph updates is further amplified if the shared data segment undergoes modifications, as shown in Figure 5. The percentage of the global index requiring recalculation continues to rise with an increasing proportion of graph updates. 3) Diverse data distribution patterns across different datasets make it challenging to select an optimal number of index vertices to balance index coverage and associated overhead. Figure 6 illustrates significant variations in inherent costs and the effectiveness of shared computations, even when the same number of index vertices is selected across different datasets.

In conclusion, the global indexing mechanism itself incurs substantial costs, and these costs fluctuate significantly across different datasets. Existing systems often conservatively opt for a limited number of global index vertices (e.g., 16 vertices in Tripoline[2] and SGraph[1]) to avoid introducing expensive overhead. However, this choice also restricts the coverage of the global index, hindering efficient computation sharing.

C. Our Motivation

Inspired by the observations mentioned above, we derive the following insights:

Data access similarity: A significant portion of traversal paths in different query tasks displays substantial overlap, indicating a similarity in data access patterns(As shown in Figure 1). However, existing solutions fall short in concurrently supporting point-to-point queries and facilitating data sharing among tasks, resulting in redundant data access. This observation motivates the development of an efficient, fine-grained data-sharing mechanism. By allowing various tasks to share access to the same data at different times, this mechanism aims to reduce data access costs and enhance the throughput of concurrent queries.

Computational similarity: The core subgraph, consisting of hot vertices and paths in the original graph, is frequently traversed by various tasks, highlighting computational similarity among different query tasks(As shown in Figure 2). Existing graph traversal systems either neglect computational similarity[3] or employ a costly global indexing mechanism[1][2], limiting the effectiveness of computational sharing. This observation inspires the creation of a lightweight, dual-layered indexing mechanism to share computational results of overlapping hot paths across different query tasks.

III. Overview Of GraphCPP

A. System Architecture

To enhance the efficiency of concurrent point-to-point queries, we introduce GraphCPP, a meticulously designed,

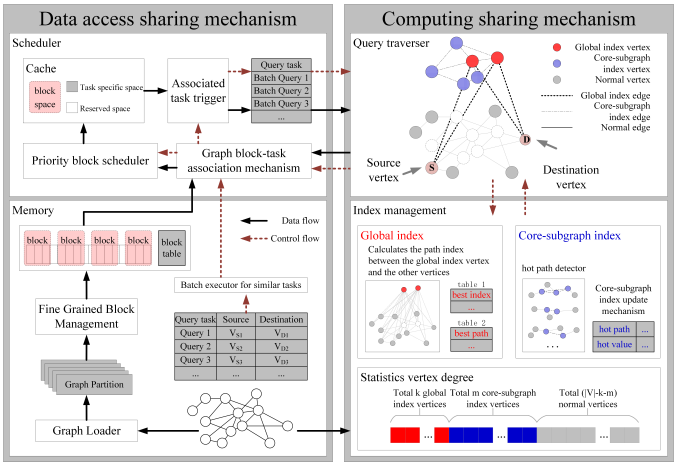

Fig. 7. System architecture diagram

data-driven system tailored for concurrent query tasks. Developed after a thorough examination of the computational intricacies inherent in such queries, GraphCPP aims to achieve data and computation sharing among concurrent point-to-point tasks. As depicted in the figure above, GraphCPP incorporates an efficient data-driven caching execution mechanism, leveraging data similarity among concurrent tasks to facilitate shared access to overlapping data. Additionally, it introduces a dual-level computation sharing mechanism, enabling computations for identical hot vertices and paths to be shared across different query tasks. Furthermore, GraphCPP employs path prediction for various queries, promoting the bulk execution of similar queries with overlapping paths and maximizing the utilization of data similarity.

Data Access Sharing Mechanism. This mechanism aims to achieve precise sharing of graph structure data by leveraging data access similarities among concurrent tasks. The process unfolds as follows: initially, similar to other distributed graph computing systems[42][56], it partitions the original graph data into coarse-grained segments, distributing them for parallel processing across different machines. Subsequently, a fine-grained chunk manager further divides these coarse-grained segments into fine-grained graph chunks. The next step involves establishing connections between query tasks and graph data chunks. Specifically, if a query task q_i has active vertices in a graph chunk b_i, a relationship between q_i and b_i is established. Following this, a graph chunk priority scheduling mechanism prioritizes graph chunks with a higher number of associated tasks for loading into the Last Level Cache (LLC). Consequently, the system filters out all query tasks linked to graph chunks present in the LLC, allowing them to be executed in batches on the shared graph chunks.

Computation Sharing Mechanism. GraphCPP optimizes shared path computation by leveraging dual-level index information (global index and core subgraph index) for concurrent query tasks. The process consists of three phases: 1) Preprocessing Phase: Gather degree information for all vertices in the original graph data. Sort vertices in descending order based on degrees. Select vertices ranked from 1 to k as global vertices and vertices ranked from $k+1$ to $k+m$

as core subgraph vertices. 2) Index Building Phase: Execute a point-to-all query algorithm before queries start. Compute the best path values from global vertices to all vertices in the graph (e.g., SSSP algorithm for PPSP tasks). For directed graphs, calculate outbound and inbound path values for global vertices; for undirected graphs, only one calculation is needed. Dynamically maintain the core subgraph through a runtime method. The core subgraph doesn't need precomputation for hot paths but explores them dynamically after each query, adding them to the core subgraph structure. 3) Computation Sharing Phase: Global vertices, acting as intermediaries for numerous paths, are used at the beginning of a query to compute the path value for a reachable path. This path may not be the best path but serves as a reference for pruning queries. In pruning traversal based on upper and lower bounds, the global index estimates path values for query paths earlier, enabling earlier pruning of unsuitable paths. This constitutes the first level of computation sharing. The core subgraph, maintaining best path values between hot vertices, acts as a highway network between query tasks. When a query task traverses hot vertices in the core subgraph, it connects to the highway, allowing quick traversal from entrance to exit vertices without recalculating path values. This constitutes the second level of computation sharing.

B. Overall Execution Workflow

Algorithm 1 outlines the comprehensive execution process of GraphCPP in handling concurrent query tasks. We obtain the set $block_{table}$, consisting of all graph blocks, and the set Q, representing all query tasks, both specific to the current computing node. Each of these sets is allocated dedicated memory space(line 2). In the third line, we enter a loop processing phase, and queries iterate until convergence is achieved. GraphCPP invokes `ChooseNextSharingBlock` to update the association between query tasks and graph blocks and selects the graph block b_i with the highest priority (line 4). By tallying the associated blocks for each task (i.e., tasks with active vertices in the current block), we can determine all query tasks related to the current graph block b_i (line 5). Next, we load b_i into the cache and simultaneously process all associated query operations q_i (line 6). This step embodies the idea of data sharing, i.e., multiple query tasks sharing the result of a single data access. Subsequently, we invoke `GraphCPPCompute` to perform point-to-point query operations q_i on the current block (details are provided in Section (section:xx) . Iterative query task execution follows the Bulk Synchronous Parallel (BSP) model, where each iteration generates a new set of active vertices and updates to form a new query task (line 7). If the new query remains associated with the current graph block b_i, we add q_i to Q_{b_i} and return to the sixth line to continue querying on the graph block b_i(line 9). Otherwise, it signifies that the new task is not associated with the current graph block, and it needs to be saved in the query task set(line 11), at this point, the task is suspended.

The presented algorithm illustrates the overall workflow of GraphCPP. In the subsequent sections, we will delve into the detailed explanations of two optimization mechanisms: data access sharing and compute sharing.

Algorithm 1 Concurrent Point-to-Point Queries

```
1: function OVERALLWORKFLOW
2:     MALLOCBUFFERS(block_table, Q)
3:     while HAS_ACTIVE(block_table) do
4:         b_i ← CHOOSENEXTSHARINGBLOCK
5:         Q_{b_i} ← CHOOSEASSOCIATEDQUERIES(b_i)
6:         for each q_i ∈ Q_{b_i} do
7:             new_query ← GRAPHCPPCOMPUTE(q_i, b_i)
8:             if HAS_ASSOCIATED((b_i, new_query)) then
9:                 Q_{b_i}.Push(new_query)
10:            else
11:                Q.Push(new_query)
12:            end if
13:        end for
14:    end while
15: end function
```

C. Data Access Sharing Mechanism

In Section 2.2, we observed a significant overlap in graph structure data access among concurrent tasks. Under the existing processing mechanism, this overlapping data cannot be shared and utilized. However, for point-to-point query tasks on the graph, the order of data access does not affect the correctness of the results. The core idea of our data sharing mechanism is to transform the original $task \rightarrow data$ linear task scheduling order into a $data \rightarrow task$ fine-grained concurrent task scheduling order. This allows us to leverage data similarity among concurrent query tasks, amortize data access overhead, enhance cache utilization efficiency, and consequently, improve system throughput. We will address two key challenges in the following sections: 1) How to determine the shared data segments? 2) How to implement data sharing among multiple tasks? Finally, we will describe additional measures to further exploit data access similarity.

a) Determine Shared Data Segments

Determine the granularity of shared graph partitions. Distributed memory graph computing systems need to load data into the cache to improve data access efficiency. Ideally, the data of shared graph partitions should fully fit into the Last-Level Cache (LLC) to avoid frequent eviction and reloading of different parts of partitions. However, the granularity of graph partitions should not be too small, as it would increase synchronization overhead in task processing. We use the formula x to determine an appropriate size for shared graph partitions. Here, B_S represents the size of the graph structure data for the shared partition, G_S represents the size of the graph structure data for the partition it belongs to, $|V|$ is the total number of vertices on the partitioned graph, V_S represents the average space required to store state information for a vertex, N is the number of concurrent query tasks, LLC_S is the size of the LLC cache, and R_S is the size of reserved redundant space, which can be used to store data such as indexes. The two terms on the right side of the formula represent graph structure data and task-specific data, respectively (their sizes are proportional to the scale of the graph partition and the number of concurrent query tasks). The right side of the formula represents the size

of the available space per task after deducting the reserved cache space. Using this formula, we determine the maximum granularity for each shared graph partition under the premise of adapting to LLC capacity.

$$B_S + \frac{B_S}{G_S} \cdot |V| \cdot V_S \cdot N \leq LLC_S - R_S \quad (1)$$

Logical Partitioning. Once the granularity of shared graph blocks is established, GraphCPP can proceed with the logical partitioning during the graph preprocessing phase. This process involves subdividing coarse-grained graph partitions on the distributed system into finer-grained shared graph blocks. Pseudocode for partitioning graph blocks in GraphCPP is presented in algorithm 2:

Algorithm 2 Logical Partition Algorithm

```
1:  function PARTITION(P_i, block_table)
2:      block_map ← null
3:      for each e ∈ P_i do
4:          if e.src in block_map then
5:              block_map[e.src] ++
6:          else
7:              block_map[e.src] ← 1
8:          end if
9:          if block_map.size() ≥ B_S then
10:             block_table.push(block_map)
11:             block_map.clear()
12:         end if
13:     end for
14: end function
```

The logical partition function takes two parameters: one is the graph partition structure data P_i recorded in edge table format, and the other is the set of graph blocks $block_{table}$ owned by the partition (Line 1). In Line 2, we utilize a dictionary structure called $block_{map}$ to collect information about graph blocks. Its key records the source vertex ID of an edge, and its value records the number of outgoing edges corresponding to that vertex. In Line 4, GraphCPP iterates through each edge in the partition. If the edge has already been loaded into the current partition, the count of outgoing edges for that partition is incremented (Line 6). If the vertex is added to the dictionary for the first time, the count of outgoing edges for the partition is set to 1 (Line 8). After processing each edge, there is a check to determine if the current block is full (Line 11). If the block is full, the current block is added to the $block_{set}$ (Line 12), and the recorded block information is cleared (Line 13). This way, when all the data in the partition has been traversed, and each edge in the partition is assigned to a specific graph block, we obtain the collection of logically partitioned graph blocks.

b) Share Similar Data Among Multiple Tasks

Establishing the Association between Shared Blocks and Query Tasks. Through the previous steps, we have achieved fine-grained graph partitioning in a logical manner. Since this partitioning is only logical, the data remains contiguous on the physical storage medium. Therefore, it is easy to determine the partition in which a vertex is located based on its ID. During query execution, each task q_i maintains an active vertex set $Set_{act,i}$ throughout the iterative computation process. It follows the updating strategy: 1) Initially, $Set_{act,i}$ only contains the source vertex Si of the query. 2) Process the active vertices in $Set_{act,i}$ according to the flow of the point-to-point query algorithm, removing the processed vertices from the active set. 3) If a vertex's state changes in this round and it is not pruned, add the vertex to $Set_{act,i}$ for processing in the next round. We first deduce the graph block to which a vertex belongs by reverse inference of its ID and then utilize a specially designed array to store the traversed partitions for each task. Since point-to-point queries adopt a pruning-based traversal strategy, the number of active vertices in each execution round is not high. Therefore, establishing the association between query tasks and their respective blocks can be done with relatively low overhead.

Determining the Priority of Partition Scheduling. After establishing the association between query tasks and their corresponding blocks, we can tally the number of tasks associated with each block. The higher the task count, the more tasks share the block, indicating greater benefits from processing this block. Consequently, blocks with a higher task count are prioritized for placement into the Last-Level Cache (LLC).

Triggering Concurrent Execution of Associated Tasks. Having obtained the shared graph data blocks, based on the association between shared blocks and query tasks, we can infer the active query tasks. These tasks share the graph structure data in the LLC, and we execute these query tasks using a batch computing approach. As shown in Algorithm X, active tasks generate new active vertices after one round of execution. If these new active vertices remain associated with the current shared block, the query tasks continue execution. Shared blocks always remain in the LLC until all query tasks associated with them are processed, at which point they are evicted.

c) Batch Execute Similar Tasks

At any given moment, a diverse set of random query tasks exists in the task pool, each with distinct query paths. Tasks with high path overlap demonstrate efficient data sharing, while low overlap hinders this efficiency. Notably, we observe that higher task similarity correlates with increased path overlap, resulting in enhanced data access efficiency and reduced redundant synchronous iterations. Task similarity is quantified by calculating distance values between different task starting points (destination vertices). In GrapCPP, we propose a batch execution strategy that is aware of similar tasks, selecting batches of similar tasks from the task pool for execution. This approach further leverages data and computation similarity. Specifically, GraphCPP randomly selects a query task from the task pool, retrieves the starting and destination vertices of the task, and performs BFS algorithm to obtain the neighbor vertex sets Set_S for the starting vertex and Set_D for the destination vertex. Note that, considering that some central nodes may have a large number of neighbor nodes, we set an upper limit of 500 for neighbor nodes. Subsequently, it traverses the task pool, filtering out all queries with starting points in Set_S and destination points in Set_D, treating them as similar tasks to be processed concurrently. It's important to note that if the

starting or destination vertex of a query belongs to a high-degree vertex, indexing can be directly used to accelerate the query without employing regular query steps. Excluding high-degree vertices, the overhead of the k-hop SSSP itself is minimal, and the execution process can be concurrent with regular queries, making the execution cost negligible.

D. Computation Sharing Mechanism

GraphCPP achieves computation sharing through two mechanisms: the global index and the core subgraph index. The inherent overhead of the global index is substantial and directly proportional to the number of global indices. In practice, the number of global vertices is typically kept low (e.g., 16) due to this overhead. However, because of the power-law distribution, these few hot vertices act as hub nodes for various queries. As a result, for most point-to-point queries q_i, a path can be traced from the source vertex s_i through at least one global vertex to the destination vertex d_i While this path may not always be the optimal route for q_i, it serves as a valuable pruning boundary. GraphCPP utilizes the global index to rapidly identify these intermediary paths, constituting the first level of computation sharing. Additionally, the core subgraph mechanism, without preprocessing, reveals optimal paths based on existing query results, facilitating computation sharing for distinct yet overlapping hot paths. Compared to the global index, the core subgraph is lighter, allowing for a broader coverage range by increasing the number of hot vertices. Consequently, more precise pruning bounds are provided, accelerating the convergence speed of pruning queries. Algorithm 3 outlines the pseudocode for the computation sharing mechanism.

Algorithm 3 Shared Computation Algorithm

```
1: function IndexPreprocess(V, k, m)
2:     Set_global, Set_core_subgraph ← SortByDegree(V, k, m)
3:     Set_global_index ← BuildGlobalIndex(k)
4:     InitCoreSubgraph(m, Set_core_subgraph_index)
5: end function
6: function SharedComputation
7:     bound ← FirstLevelSharing(Set_global_index, Set_query)
8:     while activeVerticesCount ≠ 0 do
9:         activeVertex ← GetNextActiveVertex()
10:        if activeVertex ∈ coreSubgraph then
11:            UpdateBounds(bound,
12: SecondLevelSharing(Set_core_subgraph_index, Set_query))
13:        end if
14:        for each neighbor of activeVertex do
15:            UpdateBoundsByNeighbors(neighbor)
16:        end for
17:        activeVerticesCount ← UpdateActiveVertices()
18:    end while
19: end function
20: function MaintainCoreSubgraph(Set_query_path)
21:    Set_hot_path ← ExtractHotPath(Set_query_path)
22:    AddToCoreSubgraph(Set_hot_path)
23: end function
```

The steps for implementing computation sharing are as follows: 1) Index Preprocessing (Lines 2): After sorting the degrees of vertices, the system selects the top $k + m$ hot vertices with the highest degrees. The first k vertices serve as $global\ vertices$ (where k is user-defined), and the remaining vertices become $core\ subgraph\ vertices$. The computation of the global index is completed during preprocessing phrase. GraphCPP executes the SSSP algorithm to calculate the optimal paths (including index values and parent nodes) for k high-degree vertices to all vertices in the graph. The results are stored in an array indexed by the high-degree vertices' IDs. The core subgraph omits the precomputation process, directly reusing the computation results of each query, requiring only initialization during preprocessing. 2) Computation Sharing (Lines 6-19): $Global\ vertices$ act as pivotal nodes for query paths. Most queries have at least one path passing through global index vertices. Although this path may not be the optimal path, it provides a reliable reference for query pruning. Therefore, before executing point-to-point queries, an approximate boundary is determined using the $global\ index$, representing the first level of computation sharing. Subsequently, an iterative query algorithm is executed, continuously processing new active vertices until all vertices converge. For each active vertex, it is determined whether it belongs to the core subgraph. Initially, the core subgraph is empty and does not participate in sharing. As query tasks execute, the core subgraph gradually accumulates more hot paths. When an active vertex belongs to the core subgraph, the hot path value for the corresponding starting vertex can be directly obtained through the core subgraph, avoiding redundant computation. Additionally, the core subgraph allows the query boundary to jump directly from one hot vertex to another through the hot path, accelerating the speed of point-to-point queries. 3) Maintain the Core Subgraph (Lines 20-23): To ensure the lightweight nature of the core subgraph, hot paths are not precomputed. Instead, a subset of hot paths is explored from existing optimal paths, reusing previous computation results. Clearly, any path between any two vertices on an optimal path is also an optimal path. Therefore, with minimal overhead, identifying hot vertices from existing results and calculating results between hot vertices using a prefix sum method is sufficient. To achieve this, traversal paths and path values from the source vertex to each intermediate point need to be retained during the query process. Since point-to-point queries inherently require calculating this information, the extra overhead is minimal. Through these steps, we achieve efficient data sharing using a lightweight $core\ subgraph\ index$.

E. Update Mechanism

In practical applications, the underlying graph traversed by query tasks often undergoes dynamic changes, involving edge additions (e_{add}) and edge deletions (e_{delete}). Changes in the graph structure data can lead to errors in index values. Therefore, when dynamic updates occur in the dynamic graph, we not only need to update the graph structure information but also dynamically update the indexes. **Graph structure information update**: GraphCPP stores the out-neighbor of each vertex using an adjacency list. Therefore, we only need to modify the adjacency list of the corresponding out-neighbors

based on the source vertex information when an edge is added (or deleted). **Index update**: We adopt an incremental updating approach, sequentially updating the global index and core subgraph index, minimizing redundant computation costs during index updates.

The number of global index vertices is relatively small (k global index vertices), but it records a substantial number of index values ($k \times |V|$ global index values). Storing these index values in a dedicated array might result in excessive bulkiness, but dispersing the index information across individual vertices is a more suitable approach. Each vertex maintains two tables: $table1$ records the parent nodes on the optimal paths to k global vertices, and $table2$ records the index values of the optimal paths from this vertex to k global vertices. Incremental updates are performed on these two tables based on the type of edge update. Specifically, when an edge addition update (e_{add}) occurs, we first obtain the source vertex src, destination vertex dst, and the weight between the two points. We then sequentially check each global index vertex. If $Index_{\text{src}} + weight > Index_{\text{dst}}$, we update $parent_{\text{dst}}$ in $table1$ to src and $Index_{\text{dst}}$ in $table2$ to $Index_{\text{src}} + weight$. Otherwise, there is no need to update the index for that global vertex. In the case of an edge deletion update, we check each global index vertex and determine if $parent_{\text{dst}}$ equals src. If true, it indicates that we have deleted the original optimal path to dst, and we need to recalculate $Index_{\text{dst}}$. Similar to other incremental computation methods[29][30] updates to dst will gradually propagate outward, requiring updates for all downstream vertices on the optimal paths that pass through dst. If $parent_{\text{dst}}$ is not equal to src, there is no need to update that global index vertex.

The core subgraph index records indexes between a small number of high-degree vertices, requiring the maintenance of at most $m \times m$ index values (where m is orders of magnitude smaller than the graph's data scale). To store this information, we use an independent two-dimensional array for the core subgraph. Specifically, edge additions can introduce new shortcuts, leading to the degradation of original optimal paths into non-optimal ones. In the context of our pruning queries, although non-optimal path indexes can cause early overestimation of boundary values, the iterative process ensures that point-to-point queries still converge towards more optimal paths, ultimately reaching the correct optimal paths. Extracting the latest hot path values from these converged paths completes the update to hot paths. For edge deletion updates, GraphCPP checks if both vertices of the deleted edge appear on some hot path. If yes, the original hot paths are interrupted, rendering all affected hot paths invalid. If only one vertex or no vertices appear on some hot path, the deleted edge does not affect hot paths, and there is no need for an update. Since the core subgraph index reuses the optimal path results from each query, no separate calculation is needed, resulting in overall low overhead.

The above mechanism implements incremental maintenance of graph structure data, global indexes, and core subgraph indexes. Considering that subtle graph updates do not significantly impact the overall computation results, we temporarily store subtle graph updates ΔG until its size exceeds a preset threshold or a certain time interval is reached. Only then do we execute batch graph update operations, further reducing update costs.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

Hardware Configuration: The experiments are conducted on an 8-node cluster, each machine equipped with 2 32-core lntel Xeon platinum 8358 CPUs (each CPU has 48MB L3 cache) and 1GB memory. All nodes are interconnected through an lnfiniband network with a bandwidth of 300Gbps. The programs are compiled using gcc version 9.4.0, openMPl version 4.0.3, and with openMP enabled.

Graph Algorithms: To align with prior work[1][2][3][4], we've selected six benchmark algorithms widely applied in graph clustering, classification, and prediction. These algorithms fall into two categories: graph and undirected graph algorithms. 1) Weighted graph algorithms: Point-to-point shortest path (PPSP)[18][19][20], point-to-point widest path (PPWP)[21], and point-to-point narrowest path (PPNP)[22] are utilized to identify the shortest, widest, or narrowest paths between two vertices. They find extensive applications in social/financial networks, traffic planning, monitoring money laundering activities, and network quality analysis. 2) Unweighted graph algorithms: Breadth-first search (BFS)[23], connectivity[24][25][26], and reachability[27] are the three most common pairwise queries on unweighted graphs. They are employed to determine the shortest path between two vertices, whether two vertices are connected in an undirected graph, and whether two vertices are connected in a directed graph. These algorithms find widespread use in biconnectivity, higher-order connectivity, and advanced algorithms for graph clustering.

Graph Datasets: The graph datasets used for the aforementioned algorithms are presented in Table x. LiveJournal and Twitter-2010 belong to social network graphs, while UK-2007-05 and Gsh-2015-host represent web-crawl graphs. These datasets exhibit power-law graphs with small diameters and skewed distributions, capturing real-world graph distribution scenarios. Our experiments are based on dynamic graphs, utilizing a snapshot mechanism where graph updates are performed on an unclosed snapshot, and graph queries are executed on a closed snapshot. Unclosed snapshots are periodically transformed into closed snapshots, replacing the original snapshot.

Datasets	Vertices	Edges	Data sizes
LiveJournal[1]	4.8M	69M	526MB
Twitter-2010[2]	41.7M	1.5B	10.9GB
Gsh-2015-host[3]	68.7M	1.8B	13.4GB
UK-2007-05[4]	106M	3.74B	27.9GB

TABLE III: Dataset Information

System Comparison: We conducted a comparative analysis between GraphCPP and existing point-to-point query solutions, including PnP[3], Tripoline[2], and SGraph[1]. In Section x, as SGraph represents the state-of-the-art in point-to-point query systems, the performance of PnP-S, PnP-C,

Tripoline-S, and Tripoline-C consistently falls behind SGraph. Therefore, in the subsequent comparative analysis, we did not present the performance results of PnP and Tripoline. Since these systems are not open source, we re-implemented their mechanisms based on the Gemini distributed graph processing framework. Given that PnP, Tripoline, and SGraph do not directly support concurrent operations, we made modifications to enable them to handle multiple queries simultaneously. Systems with the "-S" suffix execute tasks sequentially, while those with the "-C" suffix handle queries concurrently, managed by the operating system.

Additionally, for a more scientifically rigorous experimental comparison, we benchmarked against the state-of-the-art general concurrent graph computing solution, Glign. When evaluating the overall performance of concurrent queries, we employed a mixed submission approach, generating an equal number of queries for algorithms such as PPsP, PPWP, PPNP, BFS, Connectivity, and Reachability. It is noteworthy that a global indexing mechanism needs to maintain a set of indices for different graph algorithms. To ensure fairness, we calculated the average global indexing cost for the six algorithms. Furthermore, random parameters were set for each query task, maintaining consistency with the baseline: the number of global indices (k) was set to 16, and the core subgraph vertices in GraphCPP were set to 128. All benchmarks were executed 10 times, and the reported experimental results represent the average values.

B. Overall Performance Comparison

Figure 8 depicts the total execution time of xx concurrent queries using different schemes. Due to significant variations in the execution times of different test cases, we normalize the execution time relative to PnP's performance. It is evident that, for all graphs, GraphCPP achieves shorter execution times (and higher throughput) compared to the other schemes. In comparison to SGraph-S, SGraph-C and Glign, GraphCPP demonstrates an average throughput improvement of approximately xx, xx, and xx times, respectively. This enhancement in throughput is accomplished by reducing data access costs and efficiently pruning the core subgraph in GraphCPP.

To provide a more in-depth analysis of performance, we further divide the total time into data access time and graph processing time. As shown in Figure 9, GraphCPP requires less time for graph data access compared to other systems, and this proportion decreases further as the graph size increases. For example, in the case of Gsh-2015-host, GraphCPP's data access time is reduced by xx times to xx times compared to other systems. Two key factors contribute to GraphCPP's efficiency: 1) identical portions of graph data required by different concurrent queries only need to load and maintain a single copy in memory, reducing memory consumption. 2) graph data blocks are prioritized and periodically loaded into the LLC based on the number of associated tasks, promoting job reuse and effectively lowering LLC miss rates, minimizing unnecessary memory data transfers. Additionally, thanks to the two-level computing sharing mechanism, GraphCPP's computation time is also lower than that of other systems.

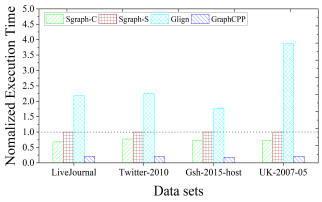

Fig. 8. Total execution time for 512 concurrent point-to-point queries using different schemes

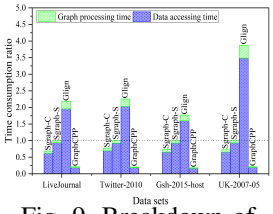

Fig. 9. Breakdown of execution times for queries of different schemes

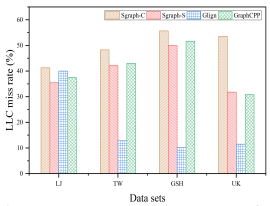

Fig. 10. LLC miss rate for 512 queries different schemes

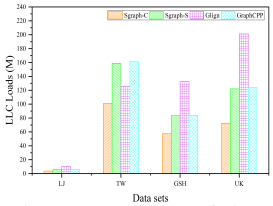

Fig. 11. Volume of data swapped into the LLC for 512 queries

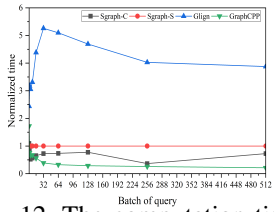

Fig. 12. The computation time for different schemes to execute queries varies with processing batches

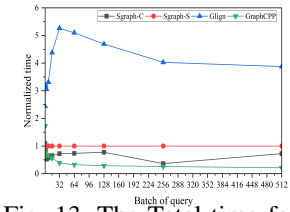

Fig. 13. The Total time for executing queries in different schemes varies with the number of concurrent queries

C. Efficiency of Data Access Sharing Mechanism

GraphCPP employs a data sharing mechanism to reduce redundant data access. To qualitatively illustrate the efficiency of this mechanism, we evaluate the LLC utilization of different systems, and the results are presented in Figure 10. Notably, GraphCPP demonstrates lower LLC miss rates compared to the other six systems. In the case of UK-2007-05, GraphCPP's LLC miss rate is only xx, while SGraph-S, SGraph-C and Glign exhibit LLC miss rates of xx, xx, and xx, respectively. This is primarily attributed to GraphCPP allowing multiple queries to share a single graph data copy in the LLC, enabling more efficient utilization of the LLC and enhancing data locality for queries.

Additionally, we track the total amount of data swapped into the LLC by these 512 queries. Generally, the concurrent execution mode (-C) tends to swap more data into the LLC compared to the sequential execution mode (-S). This is because concurrent queries lack data sharing, leading to frequent swapping of graph data in and out of the LLC and resulting in more redundant memory data transfers. As illustrated in Figure 11, GraphCPP swaps significantly less data compared to SGraph-S (xx in UK-2007-05). This reduction is attributed to GraphCPP maximizing the similarity in data access among concurrent tasks.

D. Efficiency of the Computation Sharing Mechanism

Figure 12 illustrates the relationship between system graph processing time and time. To compare the graph processing times of different systems, we normalized the processing time using SGraph-S as a reference. SGraph-C introduces a parallel execution strategy, leading to synchronization overhead, resulting in slightly longer graph processing time for individual tasks compared to the serial version. In contrast, due to the lack of specialized optimization for point-to-point queries, Glign requires a significant amount of redundant computation, making its processing time the longest. GraphCPP employs a dual-level computation sharing mechanism. In the initial phase, the core subgraph cannot participate in sharing, causing it to have a slightly higher graph processing time than SGraph-C, reverting to a regular global index. As the queries progress, new hot path values continuously join the core subgraph, gradually enhancing the computation sharing effect of GraphCPP, thereby reducing the graph processing time over time.

Global Index Overhead: As the number of global indices increases, the time required for maintaining the indices also grows. In our experiments, PnP did not incur maintenance overhead for global indices as it did not utilize them. However, Tripoline[2], SGraph[1], and GraphCPP employed similar global index mechanisms. Thus, GraphCPP's index construction overhead is comparable to other systems due to the adoption of the same number of global indices. Nevertheless, GraphCPP achieves higher throughput, allowing it to handle more queries in the same time frame, thereby distributing the global index overhead.

Core Subgraph Index Overhead: We also evaluated the impact of core subgraph indexing on GraphCPP performance. With the global index set to 16, GraphCPP-128, GraphCPP-256, and GraphCPP-without represent versions with core subgraph indices selecting 128, 256 vertices, and no core subgraph indexing, respectively. In Table x, we present the memory footprint of these versions, revealing that GraphCPP-128 and GraphCPP-256 exhibit minimal increases in memory usage compared to GraphCPP-without (all maintaining 16 global indices). This is due to the fact that core subgraph vertices store distances only to other core subgraph vertices, requiring minimal additional space compared to global indices stored for all vertices. Additionally, Table x displays the total execution times of GraphCPP-128, GraphCPP-256, and GraphCPP-without across 16 tasks. It is observed that both GraphCPP-256 and GraphCPP-128 consistently outperform GraphCPP-without, with GraphCPP-256 demonstrating superior performance to GraphCPP-128. On Friendster, the processing times for GraphCPP-256 and GraphCPP-128 are significantly lower than those for GraphCPP-without, showcasing the enhanced computational efficiency resulting from the additional core subgraph vertices, which effectively improves the computation sharing ratio and reduces redundant computational overhead.

	Core subgraph scale	Total execution time
GraphCPP-without	0	76.98
GraphCPP-128	16,384	65.46
GraphCPP-256	65,536	123.98

TABLE IV: Core Subgraph Index Overhead

Update Maintenance Overhead: The systems Tripoline, SGraph, and GraphCPP share a commonality in their global indexing mechanisms, as they all set the global index number to 16, resulting in equivalent update overhead for global indices. Simultaneously, GraphCPP introduces an additional layer of sharing through the core subgraph mechanism, incurring supplementary update costs. As described in Section xxx, when edge addition updates occur, there is no need for dedicated updates to the corresponding hot paths of core subgraphs. For edge deletion updates, only the affected hot paths need to be removed. As we do not perform dedicated computations for core subgraphs but extract hot paths from existing results, graph updates affect the accuracy of the core subgraph mechanism without incurring expensive update maintenance overhead. Furthermore, to mitigate update costs, GraphCPP conducts frequent maintenance only in the initial phase, reducing the maintenance frequency once the subgraph establishment reaches 80%. In the absence of graph mutations, previously recorded hot path values are not redundantly updated for core subgraphs.

E. Scalability of GraphCPP

Figure 16 illustrates the performance comparison between GraphCPP and six other systems under varying numbers of concurrent PPsP queries. GraphCPP exhibits superior performance improvement as the number of concurrent queries increases. For 2, 4, 8, and 16 concurrent queries, GraphCPP achieves acceleration ratios of xx, xx, xx, and xx, respectively, in comparison to SGraph-S. This improvement stems from GraphCPP's amortization, which saves more data access and storage costs with an increasing number of queries. It's important to note that with only one concurrent job, GraphCPP's fine-grained scheduling operations do not occur, resulting in minimal differences in execution time compared to other schemes. According to our tests, the block scheduling cost of GraphCPP constitutes only xx-xx% of the total execution time. Furthermore, due to resource contention, the performance of the concurrent version (-C) is considerably worse than both GraphCPP and the sequential version (-S). Therefore, a simple modification of existing graph processing systems to support concurrent tasks might not be an optimal choice.

Next, we assess the horizontal scalability of GraphCPP. To achieve this goal, we evaluate the performance of different schemes on 1, 2, 4, and 8 nodes. As shown in Figure 18, GraphCPP's performance on 8 nodes is xx-xx times that of a single machine, indicating robust scalability. Moreover, GraphCPP's scalability surpasses that of SGraph[1], PnP[3], and Tripoline[2] due to lower communication costs facilitated by data sharing and core subgraph indexing. Consequently, we believe that GraphCPP can efficiently support practical point-to-point query applications.

V. RELATED WORK

Graph Processing Systems. With the increasing prominence of graph computing as a research focus, an increasing number of diverse types of graph computing systems are being proposed. 1) Single-node in-memory graph computing

systems: Ligra[28] accelerates graph computation, especially graph traversal algorithms, by switching between push and pull computation modes. KickStarter[29] and GraphBolt[30] implement incremental processing for streaming graphs. DiGraph [x] efficiently facilitates iterative directed graph processing on the GPU. LCCG[32], DepGraph[33], and TDGraph[34] utilize topology-aware execution methods, reducing redundant computation and data access costs. 2) Single-node out-of-core graph computing systems: GraphChi[35] and X-Stream[36] achieve efficient out-of-core graph processing through sequential storage access. FlashGraph[37] employs a semi-external memory graph engine, achieving high IOPS and parallelism. GridGraph[38] adopts the Streaming-Apply approach to enhance locality and reduce I/O operations. DGraph[39] accelerates graph processing through faster state propagation. GraphM[40] adopts a data-sharing approach, optimizing throughput for concurrent systems. 3) Distributed graph computing systems: Pregel[41] introduces the Bulk Synchronous Parallel (BSP) computation model, addressing the processing needs of graphs with billions of edges in terms of performance, scalability, and fault tolerance. PowerGraph[46] proposes a vertex-cut graph partitioning approach, effectively reducing communication volume and addressing load imbalance caused by high-degree vertices. PowerLyra[49] employs a hybrid computing model, using different computation strategies for high-degree and low-degree vertices. Gemini[43] introduces a dual-mode computation engine, ensuring scalability while maintaining performance. CGraph[42] uses a data-centric Load-Trigger-Pushing (LTP) model, leveraging temporal and spatial similarities between concurrent tasks to reduce distributed system overhead. The mentioned works are intended for general graph algorithms. While the mentioned works focus on general graph algorithms, point-to-point query algorithms, concentrating on specific relationships between vertex pairs in a graph, exhibit notable distinctions. These query algorithms significantly enhance execution efficiency through pruning strategies. However, general graph computing systems have not been specifically optimized for these algorithms.

Point-to-Point Queries. Numerous studies have been conducted on point-to-point queries. On the hardware front, hub^2[50] proposed a dedicated accelerator with a hub-centric approach, accelerating the PPSP process by constraining the search scope of high-degree vertices and pruning the search process. On the software side, Quegel[51] pursued an indexing approach, enhancing query response speed by constructing a distributed static index of the graph during loading. PnP[3] took a pruning route, reducing redundant accesses and computations in point-to-point queries through a universal pruning strategy. Tripoline[2] combined both indexing and pruning, achieving incremental evaluation of queries without prior knowledge by leveraging the triangle inequality principle. SGraph[1] further optimized pruning strategies, employing upper and lower bounds to achieve sub-second latency queries on large graphs with dynamically changing data. The aforementioned efforts focus on optimizing the speed of individual point-to-point queries through mechanisms such as pruning and indexing, overlooking the substantial load posed by large-scale concurrent queries.

Concurrent Graph Computing. Numerous graph computing systems have explored concurrent graph computation, especially in the realm of single-machine memory graph processing systems. For instance, Congra[52] dynamically schedules tasks based on an offline analysis of query tasks' memory bandwidth consumption and atomic operation characteristics. This dynamic scheduling aims to enhance system throughput and resource efficiency without blocking queries due to resource contention. Krill[53] introduces an SAP model that decouples graph structure, algorithms, and attributes. It simplifies attribute data management using attribute buffers and employs graph kernel fusion to reduce memory access by treating all tasks as a cohesive unit. ForkGraph[54] proposes an efficient buffer execution model for concurrent tasks, facilitating data sharing and accelerating overall execution speed through a yield-based scheduling strategy. In the context of single-machine out-of-core graph computing systems, GraphM[40] leverages the temporal and spatial locality of concurrent tasks for data sharing in concurrent graph computation. For distributed graph computing systems, Seraph[55] suggests decoupling graph structure data from task-specific data to enable concurrent tasks to share common graph structure data. MultiLyra[56] and BEAD[57] distribute communication costs among cluster computing nodes through graph and boundary sharing, supporting efficient batch query evaluations. CGraph[42] serves as the distributed version of GraphM[40]. These approaches primarily focus on optimizing data access sharing for concurrent tasks, yet they overlook computation optimization between concurrent tasks and lack specialized optimization for point-to-point queries.

VI. CONCLUSION

Current point-to-point query systems predominantly concentrate on optimizing the speed of individual queries, overlooking the potential for throughput optimization in concurrent scenarios. This paper identifies significant data access and computation similarities among concurrent point-to-point query tasks. Introducing GraphCPP, a data-driven concurrent point-to-point query system, the paper utilizes a data-driven caching execution mechanism to achieve overlapping data access sharing among concurrent queries. Simultaneously, through a two-level computation sharing mechanism, it effectively realizes computation sharing among multiple tasks. Experimental results demonstrate that GraphCPP outperforms state-of-the-art graph query systems, such as SGraph, Tripoline, and Pnp, by a factor of XXX.

VII. ACKNOWLEDGMENTS

REFERENCES

[1] H. Chen, M. Zhang, K. Yang, et al., "Achieving Sub-second Pairwise Query over Evolving Graphs," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, Vancouver, BC, Canada, 2023, pp. 1–15.

[2] X. Jiang, C. Xu, X. Yin, et al., "Tripoline: generalized incremental graph processing via graph triangle inequality," in *Proc. Eur. Conf. Comput. Syst.*, United Kingdom, 2021, pp. 17–32.

[3] C. Xu, K. Vora, R. Gupta, "Pnp: Pruning and prediction for point-to-point iterative graph analytics," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, Providence, RI, USA, 2019, pp. 587–600.

[4] X. Yin, Z. Zhao, and R. Gupta, "Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing," in *Proc. ACM Int. Conf. Architectural Support Program. Languages Operating Syst.*, Vancouver, BC, Canada, 2022, pp. 78–92.

[5] "Alibaba Group," https://www.alibabagroup.com/document-1595215205757878272, 2023.

[6] "Google Maps," https://www.alibabagroup, 2023.

[7] "Facebook," www.facebook.com, 2023.

[8] "Alipay," www.alipay.com, 2019.

[9] "Cagis," http://www.cagis.org.cn, 2021.

[10] "Baidu Baike," https://baike.baidu.com, 2023.

[11] "A Map," https://ditu.amap.com, 2023.

[12] "Tencent Maps," https://map.qq.com, 2023.

[13] "Petal Maps," https://www.petalmaps.com, 2023.

[14] "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html, 2020.

[15] H. Kwak, C. Lee, H. Park, et al., "What is Twitter, a social network or a news media?" in *Proc. Int. Conf. World Wide Web*, Raleigh, NC, USA, 2020, pp. 591–600.

[16] P. Boldi, A. Marino and M. Santini, et al., "BUbiNG: Massive Crawling for the Masses," *ACM Trans. Web*, vol. 12, no. 2, pp. 1–26, 2018.

[17] P. Boldi, M. Santini and S. Vigna, "A large time-aware web graph," *ACM SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.

[18] H. Wang, G. Li, H. Hu, et al., "R3: a real-time route recommendation system," *Proc. VLDB Endowment*, vol. 7,no. 13, pp. 1549–1552, 2014.

[19] Q. Guo, F. Zhuang, C. Qin, et al., "A Survey on Knowledge Graph-Based Recommender Systems," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 8, pp. 3549–3568, 2020.

[20] K. Joseph, H. Jiang, "Content based News Recommendation via Shortest Entity Distance over Knowledge Graphs," in *Companion Proc. 2019 World Wide Web Conf.*, San Francisco, CA, USA, 2019, pp. 690–699.

[21] M. Pollack, "The maximum capacity through a network," *Operations Res.*, Vol. 8, no. 5, pp. 733–736, 1960.

[22] O. Berman, G. Handler, "Optimal Minimax Path of a Single Service Unit on a Network to Nonservice Destinations," *Transp. Sci.*, Vol. 21, no. 2, pp. 115–122, 1987.

[23] S. Schaeffer, "Graph clustering," *Comput. Sci. Rev.*, Vol. 1, no. 1, pp. 27–64, 2020.

[24] E. Cohen, E. Halperin, H. Kaplan, et al., "Reachability and Distance Queries via 2-Hop Labels," *SIAM J. Comput.*, Vol. 32, no. 5, pp. 1338–1355, 2003.

[25] L. Roditty, U. Zwick, "Improved Dynamic Reachability Algorithms for Directed Graphs," *SIAM J. Comput.*, Vol. 37, no. 5, pp. 1455–1471, 2008.

[26] A. Zhu, W. Lin, S. Wang, et al., "Reachability Queries on Large Dynamic Graphs: A Total Order Approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Snowbird, UT, USA, 2014, pp.1323–1334.

[27] L. Dhulipala, C. Hong, J. Shun, "A framework for static and incremental parallel graph connectivity algorithms," *arXiv preprint arXiv:2008.03909*, 2020.

[28] J. Shun, J. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. Symp. Princ. Pract. Parallel Program.*, Shenzhen, China, 2013, pp. 135–146.

[29] K. Vora, R. Gupta, G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, New York, NY, USA, 2017, pp. 237–251.

[30] M. Mariappan, K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. EuroSys Conf.*, Dresden , Germany, 2019, pp. 1–16.

[31] Y. Zhang, X. Liao, H. Jin, et al., "DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs," in *Proc. Int. Cinf. Architectural Support Program. Languages and Operating Syst.*, Providence, RI, USA, 2019, pp. 601–614.

[32] J. Zhao, Y. Zhang, X. Liao, et al., "LCCG: A Locality-Centric Hardware Accelerator for High Throughput of Concurrent Graph Processing," in *Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, 2021, pp. 1–14.

[33] Y. Zhang, X. Liao, H. Jin, et al., "DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing," in *IEEE Int. Symp. High-Perform. Comput. Architecture* , Seoul, Korea, 2021, pp. 371–384.

[34] J. Zhao, Y. Yang, Y. Zhang, et al., "TDGraph: a topology-driven accelerator for high-performance streaming graph processing," in *Proc. Annu. Int. Symp. Comput. Architecture*, New York, NY, USA, 2022, pp. 116–129.

[35] A. Kyrola, G. Blelloch, C. Guestrin, "GraphChi:Large-Scale Graph Computation on Just a PC," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, Hollywood, CA, USA, 2012, pp. 31–46.

[36] A. Roy, I. Mihailovic, W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. ACM Symp. Operating Syst. Princ.*, Farminton, Pennsylvania, USA, 2013, pp. 472–488.

[37] D. Zheng, M. Mhembere, R. Burns, et al., "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs," in *USENIX Conf. File Storage Technologies*, Santa Clara, CA, USA, 2015, pp. 45–58.

[38] X. Zhu, W. Han, W. Chen, "GridGraph:Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning," in *USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, 2015, pp. 375–386.

[39] Y. Zhang, X. Liao, X. Shi, et al., "Efficient Disk-Based Directed Graph Processing: A Strongly Connected Component Approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 830–842, 2018.

[40] J. Zhao, Y. Zhang, X. Liao, et al., "GraphM: an efficient storage system for high throughput of concurrent graph processing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage and Anal.*, Denver, Colorado, USA, 2019, pp. 1–14.

[41] G. Malewicz, M. Austern, A. Bik, et al., "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2010, pp. 135–146.

[42] Y. Zhang, J. Zhao, X. Liao, et al., "CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing," in *USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2018, pp. 441–452.

[43] X. Zhu, W. Chen, W. Zheng, et al., "Gemini: A Computation-Centric Distributed Graph Processing System," in *USENIX Symp. Operating Syst. Des. Implementation*, Savannah, GA, USA, 2016, pp. 301–316.

[44] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Proc. Hadoop Summit*, pp. 5–9, 2011.

[45] Y. Low, J. Gonzalez, A. Kyrola, et al., "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2019.

[46] J. Gonzalez, Y. Low, H. Gu, et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *USENIX Symp. Operating Syst. Des. Implementation*, Hollywood, CA, USA, 2012, pp. 17–30.

[47] J. Gonzalez, R. Xin, A. Dave, et al., "GraphX: Graph Processing in a Distributed Dataflow Framework," in *USENIX Symp. Operating Syst. Des. Implementation*, Broomfield, CO, USA, 2014, pp. 599–613.

[48] C. Xie, R. Chen, H. Guan, et al., "Sync or async: Time to fuse for distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 194–204, 2015.

[49] R. Chen, J. Shi, Y. Chen, et al., "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, no. 3, pp. 1–39, 2019.

[50] R. Jin, N. Ruan, B. You, et al., "Hub-accelerator: Fast and exact shortest path computation in large social networks," *arXiv preprint arXiv:1305.0507*, 2013.

[51] Q. Zhang, D. Yan and J. Cheng, "Quegel: A general-purpose system for querying big graphs," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2016, pp. 2189–2192.

[52] P. Pan and C. Li, "Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines," in *IEEE Int. Conf. Comput. Des.*, Boston, MA, USA, 2017, pp. 217–224.

[53] H. Chen, M. Shen, N. Xiao, et al., "Krill: a compiler and runtime system for concurrent graph processing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage and Anal.*, New York, NY, USA, 2021, pp. 1–16.

[54] S. Lu, S. Sun, J. Paul, et al., "Cache-efficient fork-processing patterns on large graphs," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2021, pp. 1208–1221.

[55] J. Xue, Z. Yang, Z. Qu, et al., "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proc. Int. Symp. High-perform. Parallel Distrib. Comput.*, Vancouver, BC, Canada, 2014, pp. 227–238.

[56] A. Mazloumi, X. Jiang, R. Gupta, "Multilyra: Scalable distributed evaluation of batches of iterative graph queries," in *IEEE Int. Conf. Big Data*, Los Angeles, CA, USA, 2019, pp. 349–358.

[57] A. Mazloumi, C. Xu, Z. Zhao, et al., "BEAD: Batched Evaluation of Iterative Graph Queries with Evolving Analytics Demands," in *IEEE Int. Conf. Big Data*, 2020, pp. 461–468.