GraphCPP: A Data-Driven System for Concurrent Point-to-Point Queries in Dynamic Graphs

IEEE Publication Technology Department

***Abstract*—With the widespread adoption of graph processing techniques in fields such as map navigation and network analysis, the demand for high throughput in executing numerous concurrent point-to-point query tasks on the same underlying graph has surged. However, existing graph query systems have primarily focused on optimizing the speed of individual point-to-point queries. When it comes to concurrent graph computations, these systems face challenges due to redundant data access overhead and computational costs.**

We've observed that, owing to the power-law distribution characteristic of graph data, traversal paths of different point-to-point queries often overlap on short segments consisting of a small number of high-degree vertices. This overlap highlights the data similarity within concurrent query tasks, serving as inspiration for our proposal of a data-driven concurrent point-to-point query system, GraphCPP. It employs a "task-graph block association mechanism" to link query tasks with specific graph data blocks, thus promoting data sharing among concurrent tasks through fine-grained graph block scheduling, ultimately enhancing data access efficiency. Additionally, it utilizes a "core subgraph precomputation strategy" to precalculate index values among highly connected vertices in the graph. This precomputation allows for the swift determination of index values for frequently shared path segments upon query initiation, thereby facilitating computational sharing and expediting query result convergence. Furthermore, during scheduling, we employ a "query path similarity prediction strategy" to group and select similar tasks from the task pool, efficiently capitalizing on data similarity within concurrent point-to-point query tasks. We compare GraphCPP with state-of-the-art point-to-point query systems, including SGraph[x], Tripoline[x], and Pnp[x]. Experimental results demonstrate that GraphCPP improves the efficiency of concurrent point-to-point queries by a factor of xxxx

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

I. Introduction

POINT-TO-POINT query tasks on graphs refer to the exploration of a specific relationship between two objects utilizing the graph as a universal data structure. Unlike traditional graph query methods, point-to-point queries on graphs specifically analyze the associations or paths between two specific vertices, without the need to consider complex queries involving the entire graph or its large-scale subsets. This targeted querying strategy endows point-to-point queries with significant optimization potential. For certain versions of monotonic graph query algorithms, such as Point-to-Point Shortest Path for SSSP (PPSP), Point-to-Point Widest Path for SSWP (PPWP), and Point-to-Point Narrowest Path for SSNP (PPNP), specific path attributes between two vertices can be accurately determined without the need for or with minimal querying and processing of unrelated other vertices or edges. Due to the efficiency of point-to-point queries in graph analysis, it has found extensive practical applications in various fields. For instance, in logistics and transportation, finding the shortest path between two locations; In social network analysis, recommending potential friends to users by examining the relationship chain between two users; In financial risk analysis, analyzing how risks propagate from one entity to another; These popular applications have raised the demand for executing large-scale concurrent point-to-point queries on the same underlying graph.

However, existing solutions for point-to-point queries have focused on accelerating the efficiency of individual queries, overlooking optimization for concurrent queries. To achieve concurrent point-to-point queries, the following two challenges need to be addressed.

Firstly, share data access between different query tasks. There exists significant overlap in the traversal paths of different query tasks. However, under the existing execution paradigm, data isolation between concurrent tasks prevents the sharing of overlapping data, resulting in redundant data access. Additionally, different tasks exhibit varying access sequences for the same graph structure data, further complicating the facilitation of data sharing.

Secondly, hare calculation between different query tasks. Graph data often adheres to a power-law distribution, where segments formed by a small number of high-degree vertices frequently appear in the best paths of different queries. Due to the abundance of neighboring vertices surrounding high-degree vertices, repeated traversals by different tasks often lead to an explosive growth in computational costs. Some existing systems have attempted to employ global indexes for computational sharing, incurring substantial costs in computation, storage, and updates. This approach limits the shared coverage range and precision of computational sharing.

In response to the previously mentioned challenges, we introduce GraphCPP, a data-driven system tailored for concurrent point-to-point queries on dynamic graphs. To address the issue of data sharing among concurrent tasks, we present a data-driven caching execution mechanism that shifts from the conventional "task→data" scheduling approach to a "data→task" strategy. This change allows for concurrent

Manuscript created October, 2020; This work was developed by the IEEE Publication Technology Department.

access to graph structure data across multiple tasks. Under this execution paradigm, GraphCPP initially determines the order of data scheduling, followed by the division of graph structure data into fine-grained blocks at the LLC level. Subsequently, it associates each query task with the relevant graph block based on the block where the active vertex set of the task resides. As the active vertices change in each round, the number of associated tasks for shared blocks is updated accordingly, with blocks having more associated tasks being given higher scheduling priority. To implement the "data→task" scheduling approach, GraphCPP utilizes an associated task triggering mechanism. It prioritizes the loading of graph blocks into the LLC and utilizes task-data association information obtained in each round to trigger batch task execution associated with the current block, enhancing efficient access to shared data. In response to the challenge of computational sharing, GraphCPP introduces a query acceleration mechanism based on core subgraphs. It streamlines the traditional "global index," which maintains index values for all vertices, into a "core subgraph index" that exclusively maintains index values between high-degree vertices. The core subgraph effectively creates direct edges between interconnected high-degree vertices, representing the best index between them. When querying a high-degree vertex, the program can access all other high-degree vertices, similar to visiting neighboring nodes, enabling computational sharing across overlapping paths. The streamlined core subgraph index incurs significantly lower overhead than the global index, allowing for the inclusion of more high-degree vertices in the core subgraph. This expands the shared coverage range of frequently shared paths, ultimately enhancing computational sharing performance. Additionally, by predicting the traversal paths of different query tasks, we prioritize batch task execution for tasks with substantial overlap, further optimizing the performance of concurrent queries.

This paper makes the following contributions:

1) Analyzed the performance bottleneck caused by redundant data access in existing point-to-point query systems when handling concurrent point-to-point query tasks. Proposed leveraging data access similarity among concurrent query tasks to optimize concurrent task throughput.
2) Developed GraphCPP, a data-driven concurrent point-to-point query system on dynamic graphs, achieving data and computational sharing among concurrent tasks. Additionally, introduced a strategy for batch execution of similar tasks.
3) We compared GraphCPP with the state-of-the-art point-to-point query system XXXXXX. The results demonstrate XXXXXXXXXX.

II. BACKGROUND AND MOTIVATION

Existing solutions have primarily focused on accelerating the speed of individual queries. For instance, PnP employs a lower-bound-based pruning method to reduce redundant access during the query process. Tripoline maintains a daily index from the central vertex to other vertices, enabling rapid queries without prior knowledge. SGraph leverages the principle of triangular inequalities and proposes a "upper bound + lower bound" pruning method, further reducing redundant access during point-to-point query processes. However, as shown in Figure x, our statistics indicate that concurrent point-to-point queries on graphs are becoming an increasingly pressing demand. They prioritize the throughput of concurrent query tasks and are more tolerant of the speed of individual queries. As depicted in Figure x, we demonstrate that existing systems exhibit poor throughput when handling large-scale concurrent queries. This undesirable outcome arises from the substantial redundant data access between concurrent tasks. To qualitatively analyze the aforementioned issues, we conducted performance evaluations of parallel point-to-point queries on XXXXX (machine configurations) using XXXXX (existing best practices) on XXXXX (graph dataset).

This chapter is divided into three parts. We first introduce some concepts in concurrent point-to-point queries. Next, we analyze the performance bottlenecks of current point-to-point query schemes when handling concurrent tasks. Finally, we present the insights obtained from our observations and analysis.

A. Preliminaries

Definition 1: Graph. We represent a directed graph as $G = (V, E)$, where V is the set of vertices and E is the set of directed edges composed of vertices in V (edges in an undirected graph can be split into directed edges in two different directions). We use $|V|$ and $|E|$ to respectively denote the number of vertices and edges.

Definition 2: Graph Partition. We use $P_i = (V_{P_i}, E_{P_i})$ to denote the i_{th} graph partition of a directed graph, where V_{P_i} represents the set of vertices in the graph partition, and E_{P_i} is the set of directed edges composed of vertices in V_{P_i}. In a distributed system, different machine-specific graph partitions P_i are distinct. We partition the graph using edge cuts, where the same vertex may appear on different computing nodes, but there is only one primary vertex, while the others are mirror vertices.

Definition 3: Point-to-Point Query. We use $q_i = (s_i, d_i)$ to represent the query corresponding to task i. Here, s_i and d_i respectively denote the source and destination vertices of query q_i. The result value obtained by query q_i is represented as $R_{s,d}$. For different algorithms, it holds different meanings. For example, for shortest path queries, R_{ib} represents the shortest path between s_i and d_i. We use $Q = q_1, q_2, \ldots, q_{|Q|}$ to represent the set of concurrent point-to-point queries, where $|Q|$ denotes the total number of queries.

Definition 4: Index: An index records the index from one vertex to other vertices and achieves computational sharing by calculating frequently accessed paths. Global Index: We select the top k vertices with the highest degrees in the graph as index vertices h_i (where $i \in [1, k]$, and users can specify k according to their needs, typically set to 16). In this context, $d_{i,j}$ (where $v_j \in V$) represents the index from index vertex h_i to any vertex v_j in the graph. If there is no reachable path between two vertices, the value is set to an extremely high value. Similarly, $d_{j,i}$ (where $v_j \in V$) represents the index

from any vertex v_j in the graph to the index vertex h_i. In undirected graphs, $d_{i,j}$ and $d_{j,i}$ are equal. The creation of this global index is designed to meet specific requirements. Core Subgraph Index: We choose highly connected vertices h_j in the degree range (k, m) to establish a core subgraph index (where $j \in [m+1, m+k]$, and users can specify m based on their requirements, typically one order of magnitude larger than k).

Definition 5: Upper Bound and Lower Bound: In point-to-point queries, the upper bound (UB) represents the known the best index value from the source vertex to the destination vertex. The lower bound (LB) for the current vertex v to the destination vertex is a conservative estimate of the best index. The predicted LB is less than or equal to the actual the best index from vertex v to the destination vertex. According to the triangle inequality on the graph, if a path's index is greater than UB or if adding the value of LB makes it greater than UB, then this path is certainly worse than existing paths and should be pruned. The values of upper and lower bounds need to be derived with the help of an index. Essentially, they are a form of computation sharing.

Definition 6: Core Subgraph. Similar to an index, a core subgraph also identifies highly connected vertices in a graph, but it employs a lower threshold for selection, which means that more vertices can be chosen. These highly connected vertices form the core subgraph, where the edge weights between two high-degree vertices represent the index values between the two points. If two vertices are ultimately unreachable, the edge weight is set to a very large value. The key distinction between the core subgraph and a global index is that the core subgraph only maintains indices among high-degree vertices and does not store index values for reaching non-high-degree vertices.

B. Redundant Computational Costs in Concurrent Tasks

Due to the characteristics of power-law distribution in graphs, a small number of high-degree vertices are connected to the majority of edges. Therefore, as illustrated in Figure 5, even though high-degree vertices represent only a fraction of the total vertices (XX%), they appear in a significant proportion of paths (XX%). Further analysis, as shown in Figure 6, reveals that a substantial portion of the overlapping path data accessed by different tasks consists of high-degree vertices. This implies that different query tasks repetitively traverse the best paths between high-degree vertices. Within a single graph snapshot period, the query paths between high-degree vertices remain constant, making the redundant computation for them unnecessary. Additionally, since high-degree vertices have numerous outgoing and incoming edges, computing the best paths between them results in substantial computational load. Some existing solutions attempt to establish a global index to reduce redundant computation for different tasks. However, as demonstrated in Figure 7, a global index faces a trade-off between the shared coverage range of vertices in the graph and the inherent overhead of the index. Specifically, when the number of indices is low, the index covers fewer paths, failing to achieve a high level of sharing. When the number of indices increases, the associated computational, storage, and maintenance costs escalate proportionally, diminishing the benefits of the index. Given the variations in the properties of different graph datasets and the evolving scenarios of concurrent queries, determining an optimal number of indices becomes challenging. Consequently, a global index is unable to effectively resolve the issue of computational redundancy

C. Our Motivation

Based on the above observations, we have gained the following insights:

Observation 1: There is data access similarity among different tasks, and a significant portion of their traversal paths overlap. However, due to the varying times at which different tasks access overlapping data, and the fact that existing point-to-point query systems do not support data sharing among tasks, accessing overlapping data results in redundant overhead. This inspires us to develop an efficient fine-grained data sharing mechanism. By enabling different tasks to share access to the same data at different times, we aim to reduce data access overhead and improve the throughput of concurrent queries.

Observation 2: Segments of paths composed of high-degree vertices are more likely to be repeatedly traversed by different tasks. Different query paths can be visualized as distinct lines, with high-degree vertices acting as intersections of these lines, frequently appearing in various tasks. Existing global index methods incur substantial costs and often impose restrictions on the number of indexed vertices, resulting in a low percentage of shareable paths. This insight motivates us to achieve better computational sharing through lightweight index.

III. GraphCPP Overview

A. System Architecture

To enhance the execution efficiency of concurrent point-to-point queries on dynamic graphs, following a detailed examination of the computational intricacies, we propose a data-driven efficient concurrent point-to-point query system, GraphCPP, as shown in the diagram below. It employs an efficient cache execution mechanism driven by data, allowing multiple tasks to share the results of a single data load by leveraging the data similarity between concurrent tasks. Additionally, it also includes a computation-sharing mechanism based on the core subgraph. This mechanism achieves computation sharing of high-frequency overlapping paths between different tasks through global index and core subgraph index. Furthermore, it leverages path prediction to drive the batch execution of similar tasks with overlapping paths, further exploiting data similarity.

The data access sharing mechanism is responsible for partitioning the graph structure data into fine-grained blocks, selecting shared graph blocks, and triggering the execution of associated tasks in batches. First, like other distributed graph computing systems, it partitions the original graph data into coarse-grained graph partitions for parallel processing on different machines. Then, a fine-grained block manager is used

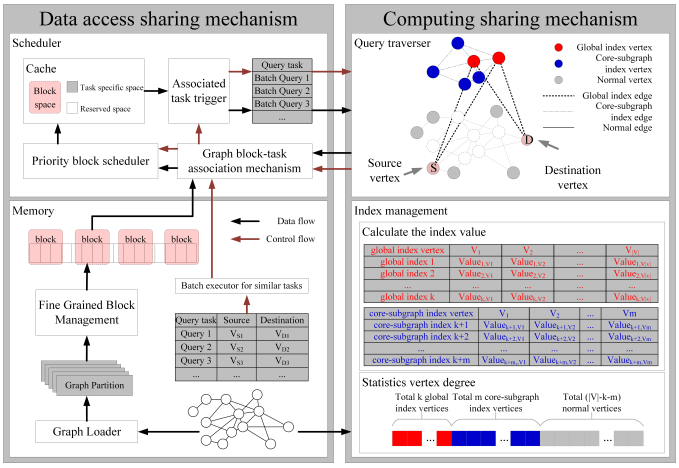

Fig. 1. System Architecture

to further divide the coarse-grained graph partitions into fine-grained graph blocks. Next, based on the partition where the active vertices of the query task are located, an association mechanism between graph blocks and tasks is established. The graph block priority scheduling mechanism will prioritize scheduling graph blocks with more associated tasks to the Last Level Cache (LLC) based on the number of associated tasks in different graph blocks. The associated task trigger, based on the active graph block information in the LLC and the association information between graph blocks and tasks, selects all tasks with association relationships to be executed in batches on the shared graph blocks.

The computation-sharing mechanism is responsible for computing index information (including global index and core subgraph index) to provide precomputed results for pruning queries. It first collects degree information for all vertices in the original graph data and sorts the vertices from largest to smallest degree. It selects vertices ranked in the range [1, k] as global index vertices and vertices ranked in the range [k+1, k+m] as core subgraph vertices. Then, it computes the indexes, where global index needs to record the index values for all vertices in the graph, and core subgraph index only needs to record the index values between core subgraph vertices. After constructing the core subgraph in the above manner, we can use it to accelerate our queries. Specifically, point-to-point queries on the graph primarily use pruning to speed up the query process. With the help of the core subgraph mechanism, we can quickly obtain the index values between high-degree vertices on the path, thereby obtaining a more accurate pruning upper bound more quickly and reducing redundant traversal of query tasks.

B. Overall Execution Workflow

We will present the overall execution flow of GraphCPP in pseudo-code. This algorithm takes two input parameters: the set B, containing all graph blocks held by the current computing node, and the set Q, containing all query tasks present on the current computing node. Initially, we allocate a dynamically-sized continuous memory space to store all query tasks (Line 1). Then, we enter a looping process as long as there are unfinished query tasks (Line 2). In this process, GraphCPP calls ChoseNextSharingBlock to update the association between query tasks and graph blocks, and select the currently highest-priority graph block, b_i. By calculating the associated blocks for each task (i.e., tasks with active vertices in the current block), we identify all query tasks related to the current graph block b_i (Line 4). Next, we load c_i into the cache and concurrently process all related query operations, q_i (Line 5). We invoke GraphCPPCompute to perform the point-to-point query operation q_i on the current block. If the query is not yet complete, we update the state of query q_i and generate new query tasks (Line 6). If the newly generated query is associated with the current graph block b_i, it is added to Q_{b_i}, and we return to Line 5 to continue querying. Otherwise, the information for the newly generated query is stored in the query task collection, and the task is suspended.

Algorithm 1 Concurrent Point-to-Point Queries on a Set of Graph Blocks Owned by a Graph Partition

1: **function** OVERALLEXECUTIONWORKFLOW(B, Q) ▷ B is the set of graph blocks, and Q is the set of query tasks
2: **while** HAS_ACTIVE(B) **do**
3: $bi \leftarrow$ ChooseNextSharingBlock()
4: $Q_{bi} \leftarrow$ ChooseAssociatedQueries(bi)
5: **for** q_i **in** Q_{bi} **do** ▷ Execute queries in Q in parallel, which is associated with block bi
6: $new\_query \leftarrow$ GraphCPPCompute(q_i, bi) ▷ The implementation function for point-to-point queries returns the active vertex set after one round of task iteration.
7: **if** HAS_ASSOCIATED((bi, new_query)) **then**
8: Q_{bi}.Push(new_query)
9: **else**
10: Q.Push(new_query)
11: **end if**
12: **end for**
13: **end while**
14: **end function**

The above algorithm demonstrates the data sharing mechanism in GraphCPP, with the GraphCPPCompute function utilizing the compute sharing mechanism. The following sections will provide a detailed explanation of these two optimization mechanisms.

C. Data Access Sharing Mechanism

In Section 2.2, we observed a significant overlap in the 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. Our data sharing mechanism essentially transforms the original "task $\rightarrow$ data" linear task scheduling order into a "data $\rightarrow$ task" fine-grained concurrent task scheduling order, thereby improving cache utilization efficiency and system throughput. Next, we will describe how

to achieve data access sharing starting from two aspects, and conclude with an additional measure that further leverages data access similarity.

Determine Shared Data Segments

1. Determine the granularity of shared graph block. Distributed memory graph computing systems need to load data into the cache to improve data access efficiency. Ideally, the data of shared graph should be able to fit entirely into the Last Level Cache (LLC), thereby avoiding the frequent swapping in and out of block parts. However, the granularity of graph blocks should not be too small, as it would increase the synchronization overhead of task processing. We employ formula 1 to determine an appropriate size for shared graph blocks. In this formula, B_S represents the size of the graph structure data for the to-be-determined shared graph block, G_S denotes the size of the graph structure data for the partition to which the block belongs, $|V|$ signifies the total number of vertices on the partition, V_S stands for the average space needed to store the status information of a vertex, N represents the number of concurrently queried tasks, LLC_S denotes the size of the LLC cache space, and R_S refers to the reserved redundant space. The two terms on the right side of formula 1 respectively represent the graph structure data and task-specific data (whose size is proportional to the scale of the graph block and the number of concurrently queried tasks). The right side of the formula indicates the size of the available space for each task after deducting the reserved cache space. Through this formula, we determine the maximum granularity of each shared graph block under the condition of accommodating the LLC capacity.

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

2. 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

```
function PARTITION(P_i, B)        ▷ B is the set of graph blocks owned by graph partition P_i.
    block_table ← null
    for each e ∈ P_i do           ▷ e is an edge in partition P_i
        if e.src in block_table then
            block_table[e.src] ++
        else
            block_table[e.src] ← 1
        end if
        if block_table.size() ≥ B_S then
            B.push(block_table)
            block_table.clear()
        end if
    end for
end function
```

Logical Partitioning Function takes two parameters: one is the graph partition structure data P_i recorded in edge table format, and the other is the collection B of graph blocks owned by this partition. We utilize a dictionary structure called $block\_table$ to aggregate information about the graph blocks, where the keys record the source vertex IDs of the edges, and values record the number of outgoing edges for each vertex. Iterate through each edge in the partition. If the edge has already been loaded into the current partition, increment the corresponding count of outgoing edges for that partition. If the vertex is added to the block dictionary for the first time, set the count of outgoing edges for the partition to 1. After processing each edge, check if the current block is full. If so, add the current block to $block\_set$. This way, after traversing all the data in the partition, every edge in the partition is assigned to a specific graph block, resulting in a set of logically partitioned graph blocks.

Share Similar Data Among Multiple Tasks

1. Establishing the Association between Shared Blocks and Query Tasks. Through the previous steps, we have achieved fine-grained graph partitioning using a logical approach. Since this is only a logical partitioning, the data remains contiguous on the physical storage medium. Hence, it is easy to determine the partition a vertex belongs to based on its ID. During the execution of a query, each task q_i maintains a set of active vertices, Set_{act_i}, following the update policy outlined below: a. Initially, $Set_(act, i)$ contains only the source vertex S_i of the query; b. The active vertices in $Set_(act, i)$ are processed according to the point-to-point query algorithm, and the processed vertices are removed from the set of active vertices; c. If a vertex's state is changed in this round and it is not pruned, the vertex is added to $Set_(act, i)$ and awaits processing in the next round; We first deduce the graph block in which the vertex is located based on its ID, and then use a specially designed array to store the partitions traversed by each task. Since point-to-point queries employ a pruning-based traversal strategy, the number of active vertices in each round is not large. Therefore, it is possible to establish the association between query tasks and their corresponding blocks with relatively low overhead.

2. Determining the Priority of Partition Scheduling. Once the association between query tasks and corresponding blocks is established, we can get the number of tasks associated with each block. The more tasks associated with a block, the greater the benefits it brings. In this scenario, the block is prioritized for loading into the Last Level Cache (LLC).

3. Triggering Concurrent Execution of Associated Tasks. Having obtained the shared graph data blocks, active query tasks associated with the LLC-resident graph structural data can be deduced. These tasks are executed in batches. As illustrated in algorithm 1, after one round of execution, active tasks generate new active vertices. If these new active vertices are still associated with the current shared block, the query tasks continue execution. The shared block remains in the LLC until all associated query tasks have been processed before it is evicted.

Batch Execute Similar Tasks

Different query tasks arrive randomly, and they have sig-

nificantly different traversal paths. We observed that when the similarity between two tasks is low, the proportion of overlapping paths between them decreases, and there may even be no overlap at all. However, if the starting and target vertices of two queries are both in adjacent graph data blocks, their traversal paths during the query process are likely to be close. To address this, we propose a batch execution strategy that is aware of similar tasks. It selects similar tasks from the task pool for batch execution, further leveraging data similarity.

Specifically, GraphCPP first randomly selects a query task from the task pool, obtaining the starting and target vertices of the task. It then performs k-hop SSSP to get the neighbor vertex sets, Set_S, for the starting vertex, and Set_D, for the target vertex (the value of k is determined by the user and is set to 3 by default). Next, it traverses the task pool, filtering out all queries with starting points in Set_S and target points in Set_D. These queries are processed concurrently as similar tasks. It's worth noting that if the starting or target vertex of a query belongs to a high-degree vertex, we can directly use an index to accelerate the query process, bypassing the 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 normal queries, with negligible cost.

D. Computation Sharing Mechanism

Tripoline initially introduced the concept of a global index, utilizing idle computational resources to maintain index values from high-degree vertices to other vertices, thus enabling the sharing of these high-degree vertex index values across different query tasks. However, the global index mechanism exhibits the following shortcomings: Shortcoming 1: The global index necessitates the recording of index values between high-degree vertices and all other vertices. When the graph's scale is extremely large, the computational and storage costs of establishing the index become substantial. Shortcoming 2: In point-to-point queries on streaming graphs, each round of graph updates introduces new edges and edge deletions. The global index requires dynamic updates of the index relationships between high-degree vertices and every vertex based on the latest graph snapshot. This implies that any update to the streaming graph impacts the index of all vertices, resulting in a significant computational overhead for maintaining the index.

In general, to better address incoming random queries at any given time, the more high-degree vertices selected, the higher the coverage of overlapping paths, leading to more effective computation sharing. However, as mentioned above, we cannot indefinitely increase the number of high-degree vertices, even if we can allocate a portion of idle computational resources to distribute the costs of calculating and maintaining the index. In response to this, this paper builds upon the global index and introduces a lightweight core subgraph index. Compared to the global index, the core subgraph index has a smaller selection threshold and a higher quantity of high-degree vertices, enabling a higher shared coverage range and providing more precise upper bound values. Additionally, it no longer maintains index values from high-degree vertices to all vertices; Instead, it only needs to maintain indices among high-degree vertices. Consequently, its overhead is significantly reduced compared to the global index. Pseudocode for core subgraph query is shown in Listing XXXX.

Algorithm 3 Core Subgraph Query Algorithm

```
1:  global_index ← BuildGlobalIndex(k)                ▷ Step 1: Calculate k Global Index
2:  core_subgraph_index ← BuildCoreSubgraphIndex(m, global_index)   ▷ Build m Core Subgraph Index
3:  function GraphCPPCompute(q, b)
4:      active_vertices ← InitializeActiveVertices(q, b)   ▷ Determine active vertices based on query task and graph block
5:      InitializeBoundsFromGlobalIndex(global_index)      ▷ Initialize bounds based on the global index
6:      while active_vertices is not empty do
7:          for vertex in active_vertices do
8:              if vertex is in core_subgraph then
9:                  UpdateBoundsByCoreVertices(vertex, core_subgraph_index)
10:             else
11:                 for nbr in GetOutgoingNeighbors(vertex) do   ▷ Traverse outgoing neighbors of vertex
12:                     UpdateBoundsByNeighbors(nbr)
13:                 end for
14:             end if
15:         end for
16:         active_vertices ← UpdateActiveVertices()
17:     end while
18: end function
```

The execution steps for achieving computation sharing are as follows: 1. Establish a Global Index (Lines 1): We employ a strategy similar to SGraph for computing the global index. After sorting the degrees of vertices, the system selects the top k vertices with the highest degrees (where the value of k is user-determined). Subsequently, an SSSP algorithm is executed to compute the shortest paths (including index values and path parent nodes) between these k high-degree vertices and all vertices in the graph. The results are stored in an array indexed by the IDs of the high-degree vertices. 2. Establish a Core Subgraph Index (Lines 2): Allowing more high-degree vertices, typically one order of magnitude larger than k, which are ranked in the top m in terms of degree, to be included in the core subgraph. As the global index vertices have already recorded the indices to reach the global index, these vertices are excluded. Additionally, once the global index is established, point-to-point queries for the best paths between points on the core subgraph can be directly computed using upper and lower bound pruning. 3. Query Acceleration (Lines 3): Perform point-to-point queries, starting by utilizing the global index to determine approximate upper and lower bounds. Subsequently, pruning queries begin. Under normal circumstances, the system traverses each outgoing edge vertex

of the current vertex, sequentially performing pruning checks on the index values of each vertex to determine the next round of active vertices. If the current query vertex belongs to the core subgraph, in addition to visiting neighboring out-edge vertices, all other high-degree vertices connected to this vertex must also be accessed. Under normal circumstances, the state propagation between these high-degree vertices may require multiple hops. With the core subgraph, the propagation between these points can be accomplished in a single step. In addition to expediting state propagation, a hidden factor is that the core subgraph is populated with high-degree vertices, making them more likely to appear on the best path between two points, thereby expediting the path discovery process. 4. Query Termination (Lines 6): Apply upper and lower bound query techniques for pruning. For unidirectional queries, starting from the source vertex, reaching the destination vertex indicates the discovery of a path. For bidirectional queries, the convergence of queries from both directions indicates the discovery of a path. If a new path value is smaller than the current upper bound, it is updated as the new upper bound. If the path value is greater than the current upper bound, it is pruned. The discovery of a path does not imply the end of iteration; It is necessary to assess the active vertices in the graph. Only when all possible paths have been attempted, the upper bound is updated to the best path value, and all vertex edge path values are greater than the current upper bound, and the number of active vertices decreases to zero, does the iteration conclude. Through the aforementioned steps, we achieve efficient data sharing using the 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}. The changing 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 indices.

Graph structure information update: GraphCPP stores the outgoing neighbor vertices of each vertex using an adjacency list. Thus, we only need to modify the adjacency list of the corresponding outgoing neighbors based on edge additions (or deletions) and the source vertex information. Index update: We adopt an incremental update approach, sequentially updating the global index and core subgraph index to minimize redundant computational overhead.

The number of global index vertices is relatively small (k global index vertices), but it records a significant number of index values ($k * |V|$ global index values). Therefore, we can store the index information on each vertex. Each vertex maintains two tables: $table_1$ records the parent nodes on the best paths to k global vertices, and $table_2$ records the index values of the best paths from this vertex to the k global vertices. Depending on the type of edge update, we perform incremental updates on these two tables. 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. Next, we sequentially check each global index vertex. If $Index_{src}$ + $weight$ ¿ $Index_{dst}$, we update $parent_{dst}$ in $table_1$ to src and $index_{dst}$ in $table_2$ to $Index_{src}$ + $weight$. Otherwise, there is no need to update the index of this global vertex. In the case of an edge deletion update, we sequentially check each global index vertex, determining whether $parent_{dst}$ equals src. If it does, it means we have removed the original best path to dst. In this case, we need to recalculate $index_{dst}$. Similar to other incremental calculation methods, updates to dst will gradually propagate outward, requiring updates to all downstream vertices of the best paths passing through dst. If $parent_{dst}$ is not equal to src, there is no need to update this global index vertex.

The core subgraph index only records a small number of indices between high-degree vertices, requiring a maximum maintenance of $m * m$ index values (m orders of magnitude smaller than the graph data scale). Therefore, we adopt a specialized storage structure. Specifically, GraphCPP utilizes the precomputed information of global index vertices, using a pruning-based point-to-point query strategy to calculate the index value of the best path between two core subgraph index vertices i and j, along with the set of vertices passed through, $set_{i,j}$. When an edge update occurs, we retrieve the starting and target points of the edge. If both are in the $set_{i,j}$, we need to recalculate the core subgraph index value between vertices i and j. Since we reuse the previous global index to implement pruning in calculating the core subgraph index, the overall overhead is relatively small.

The above mechanism achieves incremental maintenance of graph structure data, global indices, and core subgraph indices. Considering that minor graph updates will not have a significant impact on the overall computation results, we temporarily store minor graph updates ΔG until its size exceeds the preset threshold or reaches a certain time interval. Only then will we perform batch graph update operations, further reducing the update overhead.

IV. EXPERIMENTAL EVALUATION

V. RELATED WORK

Point-to-Point Queries: Existing work has conducted extensive research on point-to-point queries. For instance, hub2 [x] proposed a hub-centric specialized accelerator, which contends that vertices with a large number of connections, i.e., hubs, expand the search space, making best path calculations exceptionally challenging. It introduced the hub-Network concept to confine the search scope of hub nodes. The online pruning of hub search process was achieved using the hub2-Labeling method. However, due to hub2's specialization in a dedicated accelerator, its applicability is limited. PnP observed the traversal process of point-to-point queries and introduced an upper-bound-based pruning strategy, reducing unnecessary vertex traversals and providing a fresh perspective for point-to-point query research. Tripoline derived an approximate "upper bound" between two points by maintaining some "permanent vertices" in daily operations, using them as intermediaries. This approach enabled "prior-knowledge-free" upper bound

queries. SGraph further developed on the aforementioned methods, leveraging the triangle inequality principle on the graph to propose upper-bound and lower-bound pruning strategies, achieving sub-second point-to-point queries on the graph. However, these systems mainly focus on optimizing the speed of individual point-to-point queries, overlooking the severe load of large-scale concurrent queries.

Concurrent Graph Computing: Numerous graph computing systems have explored concurrent computing. GraphM pointed out the "data access similarity" among concurrent graph computing tasks and proposed a data-centric scheduling strategy to facilitate data sharing between multiple tasks, thereby enhancing the throughput of concurrent graph computing. However, GraphM is a single-machine out-of-core graph computing system that adopts the BSP computing model and is only applicable to static graphs. Building upon this, CGraph[x] extended the application scenarios to distributed dynamic graph computing systems. It optimized the communication mechanism and load balancing strategy for distributed scenarios. However, like GraphM, it is still an out-of-core system and is not suitable for high-load scenarios of concurrent queries, even though it can distribute the disk access cost across different subgraphs through scheduling strategies. ForkGraph efficiently conducts concurrent graph processing in memory and employs a concession-based scheduling strategy, handling only a portion of the data in each iteration to accelerate overall execution speed. However, it is a single-machine in-memory system and has not been optimized for point-to-point queries, making it unsuitable for executing concurrent point-to-point query tasks on massive datasets.

VI. CONCLUSION

This paper introduces a concurrent point-to-point query system, GraphCPP, which leverages data similarity between concurrent queries to achieve data sharing among multiple tasks. Furthermore, it employs a lightweight core subgraph index to enhance computation sharing among multiple tasks. Experimental results demonstrate that GraphCPP outperforms the state-of-the-art graph query system, SGraph, by a factor of XXX.

VII. ACKNOWLEDGMENTS

REFERENCES

[1] *Mathematics into Type*, American Mathematical Society. Online available:
[2] T.W. Chaundy, P.R. Barrett and C. Batey, *The Printing of Mathematics*, Oxford University Press. London, 1954.
[3] *The LaTeXCompanion*, by F. Mittelbach and M. Goossens
[4] *More Math into LaTeX*, by G. Grätzer
[5] *AMS-StyleGuide-online.pdf,* published by the American Mathematical Society
[6] H. Sira-Ramirez. "On the sliding mode control of nonlinear systems," *Systems & Control Letters*, vol. 19, pp. 303–312, 1992.
[7] A. Levant. "Exact differentiation of signals with unbounded higher derivatives," in *Proceedings of the 45th IEEE Conference on Decision and Control*, San Diego, California, USA, pp. 5585–5590, 2006.
[8] M. Fliess, C. Join, and H. Sira-Ramirez. "Non-linear estimation is easy," *International Journal of Modelling, Identification and Control*, vol. 4, no. 1, pp. 12–27, 2008.
[9] R. Ortega, A. Astolfi, G. Bastin, and H. Rodriguez. "Stabilization of food-chain systems using a port-controlled Hamiltonian description," in *Proceedings of the American Control Conference*, Chicago, Illinois, USA, pp. 2245–2249, 2000.

Jane Doe Biography text here without a photo.

IEEE Publications Technology Team In this paragraph you can place your educational, professional background and research and other interests.