GraphCPP: A Data-Centric System for Concurrent Point-to-Point Queries on Graph

First A. Author, *Fellow, IEEE*, Second B. Author, and Third C. Author Jr., *Member, IEEE*

[[1]](#footnote-1)

***Abstract*—With the extensive application of graph processing techniques in areas such as map navigation and network analysis, a large number of point-to-point query tasks concurrently run on a single underlying graph, posing significant challenges to existing graph query systems. Two optimization approaches are identified for concurrent point-to-point queries: 1. Accelerating the response time for individual queries; 2. Enhancing the efficiency of parallel query tasks. For the former, we introduce a high-speed core subgraph query mechanism. This mechanism selects two-tier hub vertices based on vertex degree. All hub vertices form the core subgraph, which maintains hub values based on specific rules. Leveraging the core subgraph allows for quickly determining the upper bound approximations upon receiving a query, thereby speeding up single queries. For the latter, we propose a data-centric cache processing mechanism. This divides the graph data structure into LLC-level chunks and employs priority scheduling strategies and fine-grained synchronization policies to enable data sharing among multiple tasks, optimizing cache hit rates and enhancing the throughput of concurrent queries. To our knowledge, GraphCPP is the first work optimized for concurrent point-to-point query scenarios. When compared with state-of-the-art point-to-point query systems, including SGraph[x], Tripoline[x], and Pnp[x], experiments demonstrate that GraphCPP improves the efficiency of concurrent point-to-point queries by a factor of xxxx.**

***Index Terms*—Point-to-Point Queries ,** **Concurrent Execution, Graph Processing Systems ,** **Data Access Similarity,** **Buffered Execution Model**

# I. INTRODUCTION

P

oint-to-point query tasks on a graph refer to leveraging the general data structure of a graph to explore a specific relationship between two distinct objects. Unlike traditional graph query methods, point-to-point queries on a graph specifically target the connection or pathway between two specific vertices, without concerning themselves with complex queries of the entire graph or its large subsets. This targeted query strategy offers immense optimization potential for point-to-point queries. For some monotonic graph query algorithms of point-to-point query versions, 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), they can accurately determine specific path attributes between two vertices without or with minimal query and processing of irrelevant other vertices or edges. Due to this efficiency of point-to-point queries in graph analysis, they have been widely adopted in practice across various domains. For instance: in logistics transportation, finding the shortest path between two locations; in social network analysis, recommending potential friends to users by finding the relationship chain between two users; in financial risk analysis, assessing how risks propagate from one entity to another. These popular applications have underscored the need for executing large-scale concurrent point-to-point queries on a single underlying graph.

Generally speaking, to improve the performance of concurrent point-to-point queries on a graph, two approaches can be considered: 1) Accelerate the speed of individual queries; 2) Employ efficient scheduling strategies to optimize the efficiency of parallel queries. Existing solutions for point-to-point queries focus on speeding up the efficiency of individual queries. For example: PnP employs a lower-bound pruning method to reduce redundant accesses during the querying process. Tripoline maintains daily indices from central vertices to other vertices, achieving fast queries without prior knowledge. SGraph utilizes the triangle inequality principle, introducing a "upper-bound + lower-bound" pruning method to further minimize redundant accesses during the point-to-point querying process. However, we observed that the aforementioned works do not consider scenarios with high concurrency of point-to-point queries. We demonstrate in Section 2.2 that enterprise applications often face high-concurrency query demands, and existing systems have redundant data access overheads, often resulting in significant performance bottlenecks. To address this, we introduce GraphCPP, a data-centric system for concurrent processing of point-to-point queries, specifically optimized for both individual and concurrent queries.

Optimization for Individual Point-to-Point Queries: GraphCPP introduces a rapid core subgraph query mechanism. The central idea is to utilize idle computing power to maintain indices between hub vertices, speeding up the computation process for burst query requests. Specifically, we first observed that most point-to-point query paths tend to overlap on a small subset of specific routes, which typically pass through high-degree vertices. We refer to these high-degree vertices as hub vertices and the subgraph composed of these hub vertices as the core subgraph. For each hub vertex, we maintain an index value, which records the distance between the hub vertex and all other vertices. When a new point-to-point query request arrives, the core subgraph is first consulted. From the index, we can discover a shortest path from the source vertex to the destination vertex (since the core subgraph encompasses the high-degree vertices of the entire graph, the vast majority of point-to-point queries can retrieve at least one path from the core subgraph. If no path connecting the source and target vertices is found via the core subgraph, the default upper bound is set to positive infinity). As the final path may not necessarily pass through the vertices of the core subgraph, this path might not be the globally shortest. However, it can provide us with a relatively accurate upper bound. Utilizing this, we can prune the point-to-point query process, significantly speeding up the query time. Moreover, during the bidirectional search in point-to-point queries, the core subgraph can also be leveraged to update the upper bound, optimizing the accuracy of pruning. For further details, see Chapter IV.

Optimization for Concurrent Queries: GraphCPP introduces a data-centric caching mechanism, the core idea of which is to change the scheduling order of concurrent tasks. By utilizing fine-grained scheduling, it allows shared access to overlapping graph structure data across multiple tasks. This includes the following components:1) Determining shared data portions: Within the GraphCPP framework, we've made a more detailed partition of graph structure data for each computing node, making it fit the size of LLC (Last Level Cache). Then, based on the active vertex set of the query task and its position in the graph chunk, we associate the query task with the relevant graph chunk. As the active vertices of the task change every round, the number of associated tasks that share chunks needs to be updated each round. By tracking information about associated tasks for each chunk, chunks with more associated tasks are given higher priority. They are more likely to be cached first, accelerating computations and enhancing overall efficiency;2) Implementing data sharing among multiple tasks: In traditional strategies, data accesses by tasks are independent of each other, so they cannot share data even if they process the exact same data. GraphCPP decouples the data accessed during the query process into "task-specific data" and "graph structure data". The former is related to specific query tasks and needs to be stored separately by the task. The latter pertains only to the data itself and can be shared among multiple tasks. Therefore, GraphCPP introduces a mechanism that triggers associated tasks based on the content of shared graph structure data loaded into the LLC. This mechanism initiates batch execution of associated tasks, achieving efficient access to shared data, and enhancing the throughput of concurrent tasks.

In summary, this paper makes the following contributions:

* It unveils the performance bottlenecks caused by redundant data accesses when existing graph query systems handle concurrent point-to-point query tasks. It also suggests that the similarity in data accesses between concurrent query tasks can be leveraged to optimize task throughput.
* We implemented GraphCPP, a data-centric system for processing concurrent point-to-point queries on dynamic graphs. It employs the core subgraph mechanism to optimize the speed of individual queries. Furthermore, by leveraging the similarities in data access among concurrent tasks, it accelerates the throughput of the concurrent point-to-point query system.
* We compared GraphCPP with the current state-of-the-art point-to-point query system, XXXXXX. The results indicate XXXXXXX.

# II. BACKGROUND AND MOTIVATION

Most existing point-to-point query systems are designed to optimize individual queries. However, as shown in Figure x, our statistics reveal that many real-world applications require handling of large-scale concurrent queries. These scenarios are lenient with the speed of individual queries, prioritizing the system's overall throughput instead. Yet, as depicted in Figure x, current systems exhibit poor throughput when processing large-scale concurrent queries. The unfavorable outcome arises due to the excessive redundant access to graph structure data among concurrent tasks. To qualitatively prove this point, we conducted performance evaluations of parallel point-to-point queries on XXXXXX (specific machine configuration), selecting XXXXXX (the existing best solution), on XXXXXX (the graph dataset). This chapter is divided into three sections: we first introduce some concepts in concurrent point-to-point queries; Second, we detail the pain points of the current concurrent point-to-point query schemes; Finally, we discuss the insights gathered from observations.

## A. PRELIMINARIES

Definition 1: Graph. A directed graph is represented by G=(V,E) where V denotes the set of vertices and E represents the set of directed edges formed by the vertices in V. (Edges in an undirected graph can be split into directed edges in opposite directions). |V| and |E| signify the number of vertices and edges, respectively.

Definition 2: Graph Partition. We use Pi=(VPi,EPi) to represent the ith partition of a directed graph. VPi depicts the set of vertices in the partition, and EPi is the set of directed edges formed by the vertices in VPi. In distributed systems, graph partitions Pi on different machines differ. We adopt edge-cut graph partitioning, so a single vertex might appear on multiple computing nodes, but there's only one master vertex, with others being mirror vertices.

Definition 3: Point-to-Point Query. Task i's corresponding query is represented by qi=(Si，Di), where Si and Di respectively signify the source vertex and the destination vertex of query qi. The result value obtained from query qi is RSD. Its meaning varies with different algorithms. For instance, for a shortest path query, Rib represents the shortest path between Si and Di. Q={q1,q2,…q|Q|} denotes the set of concurrent point-to-point queries, with |Q| indicating the total number of queries.

Definition 4: Upper Bound and Lower Bound. In point-to-point queries, we adopt an "upper bound + lower bound" pruning approach to further reduce redundant accesses during the process. The upper bound (UB) signifies the known shortest path distance from the source vertex to the destination vertex. As we traverse a path, if the current path distance is larger than UB, it can be ignored. If a newly discovered path distance is shorter than UB, UB is updated. The lower bound (LB) provides an aggressive shortest distance estimation from the current vertex v to the destination vertex. Derived from the triangle inequality principle of the graph, LB is always less than or equal to the actual shortest distance from vertex v to the destination. If a path's distance, combined with LB, exceeds UB, then this path is inevitably inferior to existing paths and can be pruned.

Definition 5: Hub Vertex. We use Di to denote the degree of vertex i in the graph. The system establishes two variables: threshold1 and threshold2, where threshold1>threshold2>0. Both variables are threshold values set by the user based on the system's storage resources and the scale of the graph. By traversing the vertices vi on the graph and assessing their degrees, if Di > threshold1, vertex vi is termed a hub vertex. If threshold1>Di> threshold2, vertex vi is called a sub-hub vertex. hub vertices and sub-hub vertices are collectively termed hub vertices. For hub vertices, we maintain index values concerning all other vertices (depending on the algorithm, the meaning of the hub value differs. E.g. for SSSP algorithms, the hub value signifies the shortest distance from the hub vertex to all other vertices). For sub-hub vertices, we keep index values concerning all other hub vertices.

## B. Redundant Data Access in Concurrent Tasks

The following diagram illustrates the execution of different concurrent queries on the same underlying graph. For the sake of clarity and simplicity, we have omitted the complete edges between vertices and the paths that will ultimately be pruned. Different colored arrows in the graph represent paths traversed by various query tasks. The gray circles signify ordinary vertices in the graph, while the red circles depict high-degree vertices. It's evident that paths traversed by different queries overlap, with a higher probability of overlap on paths that contain hub vertices. We have quantitatively analyzed the redundant accesses in concurrent queries. As depicted in Figure x, overlapping data accesses are prevalent in concurrent tasks, and repeated accesses to this overlapping data are considered redundant. Furthermore, as Figure x illustrates, redundant data accesses account for XXXX% of the total accesses in each query round. Due to a small number of high-degree vertices becoming popular query path candidates, they are repeatedly loaded by different queries. However, these tasks load them at different times, and even if they load the same data concurrently, existing system architectures do not support sharing this portion of data. As Figure x indicates, this data is frequently swapped in and out of the LLC (Last Level Cache), leading to a high cache miss rate, which in turn results in poor system throughput.

## C. Our Motivation

Observation 1: High-degree vertices on the graph are more likely to be traversed repetitively by different tasks. Various query paths can be visualized as distinct lines, where high-degree vertices serve as the intersection points of these lines, frequently appearing across different tasks. If we can identify these high-degree vertices, termed "hub," and establish a core subgraph among them, we can rapidly ascertain an approximate distance value for each query. While this might not always yield optimal results, using this value can enable efficient pruning during the query process, significantly hastening individual queries.

Observation 2: There is a resemblance in data access across different tasks; their traversal paths considerably overlap. This observation aligns with Observation 1. Given the asynchronous timing of overlapping data access by different tasks, and since existing point-to-point query systems don't facilitate data sharing between tasks, this leads to redundant data access overheads. This insight drove us to develop an efficient fine-grained data sharing mechanism. By enabling different tasks to share access to the same data at different times, it's possible to reduce data access overhead and enhance the throughput of concurrent queries.

# III. System Overview

To enhance the efficiency of concurrent point-to-point query execution, after meticulous research into the computational intricacies of these queries, we introduce a novel data-centric system for efficient concurrent point-to-point queries - GraphCPP. It integrates a sophisticated core subgraph querying mechanism, which maintains hub vertices to determine the upper bounds of path distances, facilitating pruning during the querying process and thus accelerating individual query speeds. More crucially, it encompasses an adept data-centric caching execution strategy. By capitalizing on data similarities among concurrent tasks, it loads graph blocks shared by multiple tasks into the LLC cache, prompting the bulk execution of associated tasks, which in turn augments cache efficiency and the throughput of the concurrent system.

Optimization of Individual Queries – Core Subgraph Query Mechanism. Before executing the query, we traverse the entire graph to gather statistics on the degree of all vertices. Based on this degree information, we filter out hub and sub-hub vertices, collectively referred to as hub vertices. The former logs the indices between the hub vertices and all vertices, while the latter keeps track of the indices between sub-hub vertices and all hub vertices. Together, these hubs constitute the core subgraph. The role of the core subgraph is to leverage known shortest distance values between vertices, thereby providing an upper bound value for the distance of unknown query vertex pairs. This value might not reflect the precise shortest distance, but it facilitates the pruning of all paths whose distances surpass it. Moreover, this upper bound can be revised as the query progresses. By employing this approach, the search space during traversal is dramatically reduced, elevating the speed of individual point-to-point queries.

Optimization for Parallel Queries – Data-Centric Cache Execution Strategy. In GraphCPP, graph partitions are further logically segmented into graph blocks the size of LLC. A mechanism for mapping associated tasks quantifies the number of tasks linked with each block. Blocks with the highest associated tasks are prioritized for loading into the LLC. An associated task trigger then initiates the bulk execution of tasks linked with the cached blocks. This data-centric approach, where a single load facilitates the sharing across multiple tasks, drastically minimizes redundant data access, bolstering the system's overall throughput.

Currently, GraphCPP supports the concurrent execution of a series of point-to-point query algorithms, such as PPSP, Viterbi, PPWP, PPNP, BFS, Reachability, Connectivity, etc. Additionally, it offers pruning capabilities for monotonic point-to-point query algorithms.

## A. System Architecture

The following describes the system architecture of GraphCPP. GraphCPP uses Gemini as its benchmark. The reason for adopting Gemini is that it currently stands as the state-of-the-art distributed in-memory graph computation engine, boasting excellent performance and programmability.

We have built upon Gemini by introducing fine-grained graph block management, associated task triggering modules, and a fine-grained data synchronization module. We retained Gemini's graph partition storage mechanism and introduced a fine-grained graph block management module that logically divides coarse-grained graph partitions into fine-grained graph blocks that can be accommodated by the LLC. This module employs a priority calculation formula, utilizing the number of associated tasks for the current block to determine its priority (the higher the number of associations, the higher the priority). The block management module schedules the highest priority blocks to cache, in the process reusing Gemini's access interface. The associated task triggering module, based on the information provided by the graph block management module, initiates the batch execution of associated tasks. Lastly, considering that while multiple tasks might access the same data block, they may do so in different orders, which could hinder data sharing. To address this, the data synchronization module adopts a fine-grained synchronization approach, enabling cache data sharing.



## B. Overall Execution Flow

The overall execution flow of GraphCPP is presented here in pseudocode format. The algorithm takes two input parameters: the collection C of all graph blocks contained in the current computing node, and the collection Q of all query tasks contained in the same node. First, a dynamic-sized contiguous memory space is allocated to store all query tasks (Line 1). The loop then continues processing as long as there are still unresolved query tasks (Line 2). GraphCPP invokes the ChooseNextSharingChunk function to select the graph block ci with the highest current priority. By evaluating the associated blocks for each task (where an associated block means the task has active vertices in the current block), we can identify all query tasks related to the current graph block ci (Line 4). Block ci is then loaded into the cache, after which all the associated query tasks qi are processed in parallel (Line 5). We use GraphCPPCompute to execute concurrent point-to-point query operations on qi in the current partition. If the query has not concluded, we update the status of query qi, resulting in a new query task (Line 6). If the newly generated query has an association with the current graph block ci, qi is added to Qci, and the process returns to Line 5 for continued querying. Otherwise, the information of the newly generated query is saved to the set of query tasks and the task is suspended.

|  |
| --- |
| Algorithm 1: Concurrent Point-to-Point Queries On Graph Chunk *C*. |
| 1: MallocBuffers( *C*, *Q* ) //*C* is the set of graph blocks, and *Q* is the set of query tasks  2: While has\_active( *C* ) do：  3: *ci* ← ChoseNextSharingChunk( )  4: *Qci* ← ChoseAssociatedQueries( *ci* )  5: Parallel\_for\_each *qi* ∈ *Qci* do: // Execute queries in *Q* in parallel, which is associated with chunk *C*  6: new\_query =GraphCPPCompute( *qi*, *ci* )  7: if(has Associated( ( *ci* , new\_query ) ):  8: *Qci*.Push( new\_query )  9: else:  10: *Q*.Push( new\_query ) |

The aforementioned algorithm illustrates the parallel query optimization in GraphCPP, which is the data-centric cache execution mechanism. Within this, the GraphCPPCompute function employs the single-query optimization—a highly efficient core subgraph query mechanism. Subsequent sections will delve deeper into these two optimization strategies.

# IV. Efficient Core Subgraph Query Mechanism

Tripoline was the first to introduce the concept of hub vertices. It speeds up single hub computation by regularly maintaining a "hub vertex index" (the distance value from the hub vertex to all other vertices, and vice versa). This provides an approximate upper bound value for any pair of vertices that might query at any time. However, the hub vertex indexing mechanism of Tripoline has the following shortcomings:

Shortcoming 1: In Tripoline's hub index design, it needs to record the hub values between the hub vertex and all other vertices. Given that graphs are often massive, the computational and storage overhead of creating this hub is considerable.

Shortcoming 2: In point-to-point queries on flow graphs, each update cycle introduces additions and deletions of edges. The hub vertex index needs dynamic updates based on the latest snapshot of the graph. Since the hub index documents the relationship of the hub vertex with every other vertex, any change in the flow graph affects all hub indices, making maintenance computationally expensive.

There is a need to depict the storage overhead of hub vertices graphically.

In general, to deal with on-the-fly random queries, the more hub vertices we have, the easier it is to find an accurate "upper bound", accelerating point-to-point query computation. However, given the aforementioned shortcomings, we cannot indefinitely increase the number of hub vertices. Even if we can offset the cost of hub computation using idle computational capacity, storage and maintenance overheads remain significant. Fewer hub vertices mean that the upper bounds we obtain are not precise, and the performance optimization is limited.

Based on these observations, we introduced a core subgraph query mechanism based on a two-level hub index, as shown in the following figure. Specifically, building upon the concept of hub vertices, we proposed a "secondary hub vertex :sub-hub." While traditional hub vertex index required recording hub values for the hub vertex with every other vertex, the index of the sub-hub vertex only needs to record hub index among the hub vertices (including both hub and sub-hub vertices). Since the number of hub vertices is much smaller than the total vertices, the storage overhead of this part is much less than that of the hub vertex. Hub vertices and sub-hub vertices together form the core subgraph. By querying using the core subgraph, we can achieve better upper bound queries with fewer hub vertices.

The execution steps of the high-speed core subgraph query mechanism are as follows: 1. Establishing Hub Vertex Index: The system first traverses all partitions, tallying vertex degrees. Those with degrees exceeding threshold 1 are designated as hub vertices, while those below threshold 1 but above threshold 2 are designated as sub-hub vertices (the computation for threshold1 and threshold2 is based on formula X). For all hub vertices, we execute a single-source shortest path algorithm to compute the shortest distance values between the hub and all other vertices. For all sub-hub vertices, we similarly calculate the shortest distance values between the sub-hub vertices and all other hub vertices. Establishing this hub vertex index is done to accommodate ad-hoc unknown queries, essentially leveraging idle computational capacity to expedite query computation. 2. Calculating Upper Bounds: The hub values of the hub vertices can swiftly determine the upper bound values for relevant queries. Specifically, upon a query's arrival, we traverse all hub vertices to retrieve the distance value Ps,i from the source point to the hub vertex and the distance value Pi,D from the hub vertex to the target point. The total path distance value via this hub vertex is Pi, where Pi = Ps,i + Pi,D. Pi might not necessarily represent the shortest path between the source and target points, but it offers a reference for our traversal, facilitating pruning. 3. Pruning Queries: Similar to SGraph's approach, GraphCPP employs a pruning strategy based on upper and lower bounds. Using Pi as the "upper bound" for pruning, all paths exceeding this upper bound in distance are pruned. Additionally, GraphCPP utilizes the triangle inequality, deducing path lower bounds through the hub index and the current upper bound Pi. Paths that, considering the lower bound, exceed Pi will also be pruned. Should a shorter path than the upper bound be discovered during traversal, the upper bound can be updated. Continually updating the upper bound sets stricter constraints, reducing the search space and accelerating the point-to-point query computation process. 4. Path Acceleration on the Core Subgraph: As observed in section 2.2, a vast majority of point-to-point query paths include multiple hub vertices (both hub and sub-hub). This observation presents a potential optimization opportunity. We deploy a bidirectional traversal algorithm; if both forward and backward traversals reach hub vertices, the shortest path value between the two hub vertices can be rapidly determined by querying the core subgraph. While this might not be the globally shortest path, it significantly refines the upper bound, thereby enhancing computation speed.



# V. Efficient Core Subgraph Query Mechanism

In Section 3.2, we observed that there's a significant overlap in the graph-structured data accessed across concurrent tasks. In the current processing mechanism, this overlapping data isn't optimally shared. For point-to-point query tasks on graphs, the order of data access doesn't affect the correctness of the results. Our cache execution model fundamentally transitions from the original linear task scheduling order to a fine-grained, data-centric scheduling order, thus enhancing cache utilization and boosting system throughput. To realize this execution model, two issues need to be addressed: 1) Determining the shared data segments, and 2) Implementing data sharing across multiple tasks. Below are the details of our implementation.

## A. Determining Shared Data Segments

Determine the granularity of shared graph data: Distributed memory systems enhance data access efficiency through caching. Ideally, the shared graph partition should be fully loadable into the Last Level Cache (LLC) to avoid frequent swapping resulting from accessing different parts of the partition. However, the granularity of the graph partition shouldn't be too fine, as this would increase the synchronization overhead. Next, we will detail how to determine the appropriate block size, considering both the graph structure data and task-specific data.

We use CS to denote the size of the shared fine-grained data block that needs to be determined, and GS to represent the size of the graph structural data on each graph partition. α denotes the proportion of the shared graph block to the partitioned graph. We use |V| to represent the total number of vertices on the partitioned graph, meaning α×|V| provides an approximation of the number of vertices in the shared block. If VS is the average space required to store the state information of a vertex, then α×∣V∣×VS signifies the maximum space needed to store task-specific data for queries on the shared block. Given that multi-core processors run tasks concurrently across various cores, the cache must store task-specific information for multiple queries. Let N be the number of threads performing parallel computations, then TS is the space required in cache to accommodate the task-specific data related to the current block's associated tasks. RS represents the size of reserved redundant space. LLCS is the size of the LLC cache. Thus, under the constraint of the following inequality, the maximum value of CS determines the size of the graph block.

Logical Partitioning. Distributed systems typically employ partitioning techniques to divide a large-scale graph into graph partitions that can fit into the memory of a single machine. On top of these memory-capacity-level graph partitions, GraphCPP further divides the graph into fine-grained graph blocks. Notably, this partitioning is a logical division, distinct from previous physical partitioning methods. Listing x shows the pseudocode for GraphCPP's graph block partitioning:

|  |
| --- |
| Algorithm: Logical Partition Algorithm. |
| 1: func. Partition(Pi , chunk\_set)  2: chunk\_edge\_num = 0  3: chunk = null  4: for each e ∈ Pi do: //e is an edge in Partition Pi  5: if e in chunk:  6: chunk[e]++  7: else:  8: chunk[e]=1  9: end if  10: chunk\_edge\_num++;  11: if chunk\_edge\_num × ≥ SC:  12: chunk\_set.push(chunk )  13: chunk\_edge\_num = 0  13: chunk.cear( )  14: end if  15: end for |

Determine the granularity of shared graph data: Distributed memory systems enhance data access efficiency through caching. Ideally, the shared graph partition should be fully loadable into the Last Level Cache (LLC) to avoid frequent swapping resulting from accessing different parts of the partition. However, the granularity of the graph partition shouldn't be too fine, as this would increase the synchronization overhead. Next, we will detail how to determine the appropriate block size, considering both the graph structure data and task-specific data.

Logical Partitioning Function receives two parameters: one representing the graph partition structure data Pi documented in edge list format, and the other chunk set, records the collection of logically divided blocks. A variable chunk\_edge\_num is then defined to account for the current partition's edge count. Another variable, chunk, is structured as a dictionary where the key is the source vertex ID, and the value represents the number of outbound edges for that vertex. During a loop that iterates through each edge in the partition, if the edge has already been loaded into the current chunk, the corresponding outbound edge count is incremented. If the vertex is being added to the chunk dictionary for the first time, the outbound edge count for the partition is set to one. After processing each edge, there's a check to determine if the current block is full. If so, it gets added to chunk\_set. By the time every data point in the partition has been processed, and each edge has been assigned to a specific graph block, we obtain a set of logically divided graph chunks.

Associating Query Tasks with their Respective Blocks: After achieving the fine-grained graph blocks through logical partitioning, it's worth noting that, although this is a logical division, the data remains contiguous in its physical storage. Therefore, we can easily determine a vertex's partition using its ID. Specifically, each query task keeps track of the set of active vertices it has traversed. By backtracking through the vertex's ID, we can deduce its block, then utilize a specially designed array to store each task's traversed partition. Given that point-to-point queries use a pruning traversal strategy, the number of active vertices in each iteration isn't extensive. Thus, it's feasible to establish an association between a query task and its block with minimal overhead.

Determining Scheduling Priority of Partitions: Once the associations between query tasks and their respective blocks are established, we can tally the number of tasks linked to each block. A greater number of tasks implies more tasks sharing that block, yielding higher benefits when the task is processed, giving such blocks higher scheduling priority.

Through these steps, we generate graph partitions available for task-sharing. With a cost-effective priority scheduling order, we load these partitions into the LLC cache. What follows now is the need for a fine-grained processing mechanism to exploit this shared data.

## B. How to Implement Data Sharing Across Multiple Tasks

Triggering Concurrent Execution of Associated Tasks: Each query task qi maintains an active vertex set Setact,i during its execution. This set abides by the following update strategy:Initially, Setact,i only contains the query source vertex Si.Following the point-to-point query algorithm's procedures, active vertices within Setact,i are processed, and once processed, they are removed from the active vertex set. If a vertex's status changes during the current round and it hasn't been pruned, that vertex is added to Setact,i for processing in the subsequent round. As previously mentioned, logical graph block divisions produce a corresponding "chunk" dictionary for each block. This dictionary documents the vertex IDs and their degree within the block. If the active vertices of task qi appear in a block's dictionary, this signifies that the task is associated with that block. Using the chunk dictionary and the active vertex set Setact,i, we can swiftly determine the concurrent execution of tasks associated with the active block loaded in the LLC, as demonstrated in Algorithm X. After one round of associated task execution, each produces a new set of active vertices. If these new active vertices continue to be linked with the current shared block, the query task proceeds. The shared block remains in the LLC until all query tasks associated with that block have been fully processed, at which point it is evicted.

同一个共享分块的任务交给一个CPU核心。多个任务用时间片轮转的方式来进行调度，为了确保负载均衡，根据任务负载来划分计算资源。

# VI. Experimental evaluation

我们的实验和SGraph一样是基于动态图的，SGraph采用了一种快照机制，图更新在未关闭快照上执行，图查询在已关闭快照执行。每隔一段时间将未关闭快照转为已关闭快照，并替换原有快照。

实验设置

预处理开销

整体性能对比

调度策略性能

是否开启索引子图对结果影响

可扩展性

# VII. related work

VIII. Conclusion

## Point-to-Point Queries

Existing research has extensively studied point-to-point queries. For instance, Hub2 [x] introduced a hub-centric dedicated accelerator. It posited that vertices with numerous connections, termed as 'hubs', expand the search space, making shortest path calculations exceedingly challenging. The hub-Network concept was introduced to restrict the search scope of hub vertices and employed the hub2-Labeling method for online pruning during the hub search. However, given Hub2's design as a specialized accelerator, its versatility is limited. PnP observed the traversal process of point-to-point queries and proposed an upper-bound based pruning strategy, which minimized unnecessary vertex traversal, thus offering a fresh perspective for point-to-point query research. Tripoline, by maintaining select "index vertices", leverages these vertices as intermediaries to deduce an approximate "upper bound" between two points, thus facilitating upper-bound queries without prior knowledge. Building on the premises of the former systems, SGraph utilizes the triangle inequality principle in graphs to introduce pruning strategies based on both upper and lower bounds, achieving sub-second point-to-point queries on graphs. Nonetheless, these systems primarily concentrate on optimizing single point-to-point query speeds, overlooking the substantial load of large-scale concurrent queries.

## B. Concurrent Graph Computation

Numerous graph computation systems have explored concurrent computing. GraphM identified the "data access similarity" among concurrent graph computation tasks and proposed a data-centric scheduling strategy. This approach enabled data sharing across multiple tasks, enhancing the throughput of concurrent graph computations. However, GraphM operates as a single-machine, out-of-core graph computation system, adopts the BSP computation model, and is only applicable to static graphs. Building on this, CGraph[x] extended its application domain to dynamic graph computation on distributed systems. It optimized communication mechanisms and load balancing strategies for distributed settings. Yet, like GraphM, it remains an out-of-core system. Even with its scheduling strategy that can distribute the overhead of disk accesses across different subgraphs, it's still not well-suited for high-load concurrent query scenarios. ForkGraph achieves efficient concurrent graph processing in-memory and adopts a yielding-based scheduling strategy, processing only portions of data each iteration, thereby accelerating the overall execution speed. However, being a single-machine in-memory system, it hasn't optimized for point-to-point queries and is unsuitable for executing concurrent point-to-point query tasks on massive data.

Ⅸ. acknowledgement

References

*Basic format for periodicals:*

J. K. Author, “Name of paper,” *Abbrev. Title of Periodical*, vol. x, no. x, pp. xxx-xxx, Abbrev. Month, year, doi: 10.1109.XXX.1234567.

*Periodicals using article numbers:*

J. K. Author, “Name of paper,” *Abbrev. Title of Periodical*, vol. x, no. x, Abbrev. Month, year, Art. no. xxxxx, doi: 10.1109.XXX.1234567.

*Examples:*

1. J. U. Duncombe, “Infrared navigation—Part I: An assessment of feasibility,” *IEEE Trans. Electron Devices*, vol. ED-11, no. 1, pp. 34–39, Jan. 1959, doi: 10.1109/TED.2016.2628402.
2. E. P. Wigner, “Theory of traveling-wave optical laser,” *Phys. Rev*., vol. 134, pp. A635–A646, Dec. 1965.
3. P. Kopyt *et al., “*Electric properties of graphene-based conductive layers from DC up to terahertz range,” *IEEE THz Sci. Technol.,* to be published, doi: 10.1109/TTHZ.2016.2544142. *(Note: If a paper is still to be published, but is available in early access, please follow ref [5]).)*
4. R. Fardel, M. Nagel, F. Nuesch, T. Lippert, and A. Wokaun, “Fabrication of organic light emitting diode pixels by laser-assisted forward transfer,” *Appl. Phys. Lett.*, vol. 91, no. 6, Aug. 2007, Art. no. 061103.
5. D. Comite and N. Pierdicca, "Decorrelation of the near-specular land scattering in bistatic radar systems," *IEEE Trans. Geosci. Remote Sens.*, early access, doi: 10.1109/TGRS.2021.3072864. (*Note: This format is used for articles in early access. The doi must be included.)*
6. H. V. Habi and H. Messer, "Recurrent neural network for rain estimation using commercial microwave links," *IEEE Trans. Geosci. Remote Sens.*, vol. 59, no. 5, pp. 3672-3681, May 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9153027

*Basic format for books:*

J. K. Author, “Title of chapter in the book,” in *Title of Published Book, x*th ed. City of Publisher, (only U.S. State), Country: Abbrev. of Publisher, year, ch. x, sec. *x*, pp. xxx–xxx*.*

*Examples:*

1. G. O. Young, “Synthetic structure of industrial plastics,” in *Plastics,* 2nd ed., vol. 3, J. Peters, Ed. New York, NY, USA: McGraw-Hill, 1964, pp. 15–64.
2. W.-K. Chen, *Linear Networks and Systems.* Belmont, CA, USA: Wadsworth, 1993, pp. 123–135.
3. Philip B. Kurland and Ralph Lerner, eds., *The Founders’ Constitution.* Chicago, IL, USA: Univ. of Chicago Press, 1987, Accessed on: Feb. 28, 2010, [Online]. Available: http://press-pubs.uchicago.edu/founders/

*Basic format for handbooks:*

*Name of Manual/Handbook, x* ed., Abbrev. Name of Co., City of Co., Abbrev. State, Country, year, pp. xxx-xxx.

*Examples:*

1. *Transmission Systems for Communications*, 3rd ed., Western Electric Co., Winston-Salem, NC, USA, 1985, pp. 44–60.
2. *Motorola Semiconductor Data Manual*, Motorola Semiconductor Products Inc., Phoenix, AZ, USA, 1989.
3. R. J. Hijmans and J. van Etten, “Raster: Geographic analysis and modeling with raster data,” R Package Version 2.0-12, Jan. 12, 2012. [Online]. Available: http://CRAN.R-project.org/package=raster

*Basic format for reports:*

J. K. Author, “Title of report,” Abbrev. Name of Co., City of Co., Abbrev. State, Country, Rep. xxx, year.

*Example:*

1. E. E. Reber, R. L. Michell, and C. J. Carter, “Oxygen absorption in the earth’s atmosphere,” Aerospace Corp., Los Angeles, CA, USA, Tech. Rep. TR-0200 (4230-46)-3, Nov. 1988.

*Basic format for conference proceedings:*

J. K. Author, “Title of paper,” in *Abbreviated Name of Conf.*, City of Conf., Abbrev. State (if given), Country, year, pp. xxxxxx*.*

*Examples:*

1. D. B. Payne and J. R. Stern, “Wavelength-switched passively coupled single-mode optical network,” in *Proc. IOOC-ECOC,* Boston, MA, USA,1985, pp. 585–590.
2. D. Ebehard and E. Voges, “Digital single sideband detection for interferometric sensors,” presented at the 2nd Int. Conf. Optical Fiber Sensors*,* Stuttgart, Germany, Jan. 2-5, 1984.
3. PROCESS Corporation, Boston, MA, USA. Intranets: Internet technologies deployed behind the firewall for corporate productivity. Presented at INET96 Annual Meeting. [Online]. Available: http://home.process.com/Intranets/wp2.htp

*Basic format for electronic documents (when available online):*

Issuing Organization. (year, month day). *Title*. [Type of medium]. Available: site/path/file

*Example:*

1. U.S. House. 102nd Congress, 1st Session. (1991, Jan. 11). *H. Con. Res. 1, Sense of the Congress on Approval of Military Action*. [Online]. Available: LEXIS Library: GENFED File: BILLS

*Basic format for patents:*

J. K. Author, “Title of patent,” U.S. Patent *x xxx xxx*, Abbrev. Month, day, year.

*Example:*

1. G. Brandli and M. Dick, “Alternating current fed power supply,” U.S. Patent 4 084 217, Nov. 4, 1978.

*Basic format**for theses (M.S.) and dissertations (Ph.D.):*

J. K. Author, “Title of thesis,” M.S. thesis, Abbrev. Dept., Abbrev. Univ., City of Univ., Abbrev. State, year.

J. K. Author, “Title of dissertation,” Ph.D. dissertation, Abbrev. Dept., Abbrev. Univ., City of Univ., Abbrev. State, year.

*Examples:*

1. J. O. Williams, “Narrow-band analyzer,” Ph.D. dissertation, Dept. Elect. Eng., Harvard Univ., Cambridge, MA, USA, 1993.
2. N. Kawasaki, “Parametric study of thermal and chemical nonequilibrium nozzle flow,” M.S. thesis, Dept. Electron. Eng., Osaka Univ., Osaka, Japan, 1993.

*Basic format for the most common types of unpublished references:*

J. K. Author, private communication, Abbrev. Month, year.

J. K. Author, “Title of paper,” unpublished.

J. K. Author, “Title of paper,” to be published.

*Examples:*

1. A. Harrison, private communication, May 1995.
2. B. Smith, “An approach to graphs of linear forms,” 2014, *arXiv:2105.02824*.
3. A. Brahms, “Representation error for real numbers in binary computer arithmetic,” IEEE Computer Group Repository, Paper R-67-85.

*Basic formats for standards:*

a) *Title of Standard*, Standard number, date.

b) *Title of Standard*, Standard number, Corporate author, location, date.

*Examples:*

1. IEEE Criteria for Class IE Electric Systems, IEEE Standard 308, 1969.
2. Letter Symbols for Quantities, ANSI Standard Y10.5-1968.

*Basic format for datasets:*

Author,  Date, Year. “Title of Dataset,” distributed by Publisher/Distributor, http://url.com (or if DOI is used, end with a period)

*Example:*

1. U.S. Department of Health and Human Services, Aug. 2013, “Treatment Episode Dataset: Discharges (TEDS-D): Concatenated, 2006 to 2009,” U.S. Department of Health and Human Services, Substance Abuse and Mental Health Services Administration, Office of Applied Studies, doi: 10.3886/ICPSR30122.v2.

*Basic format for code:*

Author,  Date published or disseminated, Year. “Complete title, including ed./vers.#,” distributed by Publisher/Distributor, http://url.com (or if DOI is used, end with a period)

*Example:*

1. T. D’Martin and S. Soares, 2019, “Code for Assessment of Markov Decision Processes in Long-Term Hydrothermal Scheduling of Single-Reservoir Systems (Version 1.0),” Code Ocean, doi: \_1.24433/CO.7212286.v1

**First A. Author** (Fellow, IEEE) and all authors may include biographies if the publication allows. Biographies are often not included in conference-related papers. Please check the Information for Authors to confirm. Author photos should be current, professional images of the head and shoulders. The first paragraph may contain a place and/or date of birth (list place, then date). Next, the author’s educational background is listed. The degrees should be listed with the type of degree in what field, which institution, city, state, and country, and year the degree was earned. The author’s major field of study should be lowercase.

The second paragraph uses the preferred third person pronoun (he, she, they, etc.) and not the author’s last name. It lists military and work experience, including summer and fellowship jobs. Job titles are capitalized. The current job must have a location; previous positions may be listed without one. Information concerning previous publications may be included. The format for listing publishers of a book within the biography is: *Title of Book* (publisher name, year) similar to a reference. Current and previous research interests end the paragraph.

The third paragraph begins with the author’s preferred title and last name (e.g., Dr. Smith, Prof. Jones, Mr. Kajor, Ms. Hunter, Mx. Riley). List any memberships in professional societies other than the IEEE. Finally, list any awards and work for IEEE committees and publications.

**Second B. Author**, photograph and biography not available at the time of publication.

**Third C. Author Jr.** (Member, IEEE), photograph and biography not available at the time of publication.

1. This paragraph of the first footnote will contain the date on which you submitted your paper for review, which is populated by IEEE. It is IEEE style to display support information, including sponsor and financial support acknowledgment, here and not in an acknowledgment section at the end of the article. For example, “This work was supported in part by the U.S. Department of Commerce under Grant 123456.” The name of the corresponding author appears after the financial information, e.g. *(Corresponding author: Second B. Author).* Here you may also indicate if authors contributed equally or if there are co-first authors.

   The next few paragraphs should contain the authors’ current affiliations, including current address and e-mail. For example, First A. Author is with the National Institute of Standards and Technology, Boulder, CO 80305 USA (e-mail: author@ boulder.nist.gov).

   Second B. Author Jr. was with Rice University, Houston, TX 77005 USA. He is now with the Department of Physics, Colorado State University, Fort Collins, CO 80523 USA (e-mail: author@lamar.colostate.edu).

   Third C. Author is with the Electrical Engineering Department, University of Colorado, Boulder, CO 80309 USA, on leave from the National Research Institute for Metals, Tsukuba 305-0047, Japan (e-mail: author@nrim.go.jp).

   Mentions of supplemental materials and animal/human rights statements can be included here.

   Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org [↑](#footnote-ref-1)