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

Yu Zhang, *Member, IEEE,* Haoyu Lu, Jianhui Yue, *Member, IEEE,* Kang Luo, Weihang Yin, Yutao Fu, Zirui He, Xiaoxuan Xu, Jiapeng Li, Jin Zhao, Xiaofei Liao, *Member, IEEE,* Hai Jin, *Fellow, IEEE*

Abstract—With the surging demand for concurrent point-to-point queries in applications like map navigation and network analysis, graph processing systems are facing an escalating challenge in maintaining high throughput. However, previous works either focus on optimizing the speed of individual point-to-point queries, neglecting the throughput of concurrent queries, or follow the concurrent execution schema of point-to-all algorithms, overlooking the optimization potential of point-to-point algorithms. Due to redundant data access and computational overhead, existing solutions exhibit poor overall throughput when executing concurrent point-to-point queries. This paper introduces GraphCPP, a novel graph processing system designed for the concurrent execution of point-to-point queries. GraphCPP achieves significant throughput improvements through two key features: 1) **Data Access Sharing**: recognizing the inherent overlap in traversal paths during concurrent queries, we propose a data-driven mechanism that facilitates shared access to redundant graph structure data, thereby amortizing the cost of data access. 2) **Computation Sharing**: noting that a few popular path segments are frequently recalculated by a substantial number of queries, GraphCPP employs a dual-level computation sharing mechanism. It accelerates the convergence of new queries by sharing previously computed values from frequently accessed paths. Based on our experiments, GraphCPP outperforms the state-of-the-art point-to-point query system, SGraph, by 3.2× on average.

Index Terms—graph process, point-to-point queries, concurrent queries, data access sharing, computation sharing

1 Introduction

Compared to point-to-all query algorithms, which consider relationships among all vertices in the entire graph, point-to-point queries focus on the specific path between source and destination vertices. This unique traversal characteristic significantly enhances query efficiency, extending the applicability of point-to-point queries in daily scenarios. Examples include optimizing logistics routes on Google Maps [1], suggesting friends through social network analysis on Facebook [2], and analyzing risk propagation in financial assessments on Alipay [3]. The increasing demand for these applications emphasizes the importance of efficiently managing high-throughput concurrent point-to-point queries on underlying graphs. While existing solutions like PnP [4], and SGraph [5] enhance the speed of individual queries through threshold-based pruning strategies, they are confined to facilitating serial queries and neglect concurrent query throughput. To meat the escalating demand for concurrent point-to-point queries, we must address the following challenges.

Queries on the underlying graph exhibit an overlap effect, where the traversal paths formed by these tasks intersect. Our experiments reveal that 72%-90% of vertices in a graph are accessed by multiple queries (Figure 3), leading to significant **redundant data access**. Additionally, natural graphs follow a power-law distribution, with a small number of vertices connecting to numerous edges. These vertices are frequently involved in the computation processes of different queries, constituting 75%-95% of the total computation (Figure 4). The repeated process for these highly connected vertices caused by different queries results in **redundant computation**. However, as illustrated in Figure 1, existing query systems adopt a task-driven approach, independently accessing and processing the necessary data without inter-query cooperation, which fails to effectively address redundant overhead. Moreover, with an increase in concurrency, the situation even exacerbates.

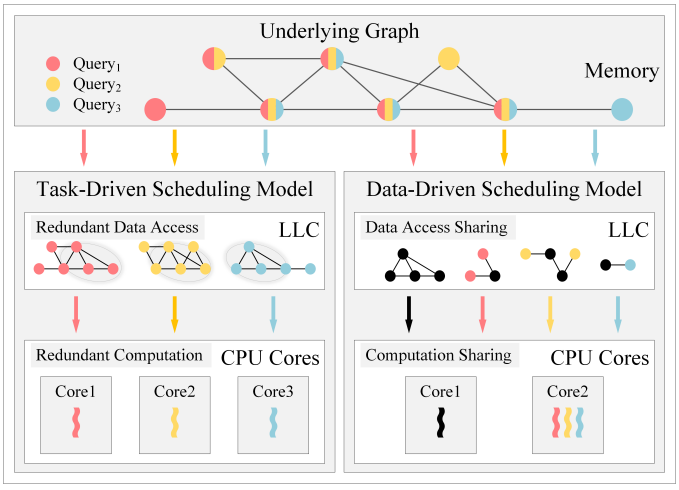


Fig. 1. The scheduling processes of task-driven and data-driven

- *Yu Zhang, Haoyu Lu, Kang Luo, Weihang Yin, Yutao Fu, Zirui He, Xiaoxuan Xu, Jiapeng Li, Jin Zhao, Xiaofei Liao, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {zhyu, hylu, luokang2000, hannyin, fuyutao, hezirui, xuxiaoxuan, goodgap, zjin, xfliao, hjin}@hust.edu.cn.*
- *Jianhui Yue is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA. E-mail: jyue@mtu.edu.*

Some solutions, such as Glign [7], attempt to optimize the throughput of general concurrent graph processing. However, Glign primarily focuses on point-to-all algorithms and lacks an appropriate mechanism to fully leverage the unique characteristics of point-to-point queries, thus failing to avoid redundant path traversals. Additionally, another solution SGraph [5], proposes a computation sharing approach. It selects high-degree vertices and performs point-to-all queries with these vertices as the starting or ending vertices, storing the query results during the preprocessing phase. In this way, tasks requiring the execution of queries associated with these high-degree vertices can benefit from avoiding recomputation. This approach reduces the query response time by pre-computing and sharing the results that may be queried. However, it introduces a substantial overhead, as its computational and storage requirements typically scale proportionally with the size of the graph, making them unaffordable for large graphs.

Driven by the increasing demand for concurrent point-to-point queries and recognizing the limitations of current solutions, we present GraphCPP—a novel data-driven system designed to optimize concurrent query throughput. GraphCPP integrates a data access sharing mechanism to facilitate concurrent queries with shared data access. It divides the data needed by query into task-specific information, which remains private to each query thread, and graph structure data, which is uniformly managed through a data-sharing mechanism. Subsequently, utilizing a priority scheduling strategy, the system loads graph blocks into the Last Level Cache (LLC). Following this, an association mechanism triggers the execution of queries related to the loaded block. These queries share the same graph structure data in LLC, eliminating redundant data access and amortizing data access overhead. Additionally, drawing inspiration from logistics networks that optimize efficiency through hub stations and high-speed connections, GraphCPP employs a dual-level computation sharing mechanism to optimize query speed. This mechanism consists of two components: the Global Vertices and Core Subgraph Mechanisms. They identify frequently accessed vertices and paths, precomputing or dynamically obtaining their query results during runtime. Such pre-calculated information can be efficiently reused by subsequent queries, significantly boosting processing speed.

This paper makes the following contributions:

1) **Analysis of Performance Bottlenecks**: we analyze the performance bottlenecks of existing systems for concurrent point-to-point queries and identify potential optimization opportunities through data access and computation sharing.
2) **GraphCPP Design**: we introduce GraphCPP, a novel data-driven graph processing system specifically designed for concurrent point-to-point queries. GraphCPP leverages redundancies in data access and computation across concurrent queries to achieve significant performance improvements.
3) **Evaluation and Performance**: we conduct comprehensive evaluations comparing GraphCPP against state-of-the-art systems on a diverse workload using real-world graphs and applications. The results demonstrate that GraphCPP surpasses SGraph by an average of 3.2×.

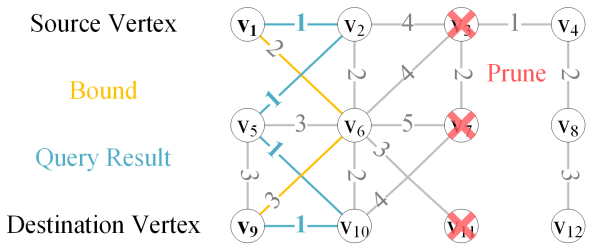
The remaining content is structured as follows: Section 2 provides background information and outlines our motivation. Section 3 details the implementation specifics of GraphCPP, followed by the experimental evaluation in Section 4. Section 5 reviews related work, and lastly, we present our conclusions in Section 6.

2 Background and Motivation

2.1 Point-to-Point Query

Application. The point-to-point query is utilized to retrieve specific **query results** between particular vertex pairs in a graph. It is typically divided into two categories: 1) Weighted graph algorithms: Point-to-point shortest path (PPSP) [13], point-to-point widest path (PPWP) [14], and point-to-point narrowest path (PPNP) [15] are employed to identify the shortest, widest, or narrowest path between two vertices, respectively. They find extensive applications in social/financial networks, traffic planning, monitoring money laundering activities, and network quality analysis. 2) Unweighted graph algorithms: Breadth-first search (BFS) [16], connectivity [17], and reachability [18] are used to determine layering (BFS) or reachability of specific vertex pairs in undirected (connectivity) or directed graphs (reachability). These algorithms are widely applied in biconnectivity, higher-order connectivity, and advanced algorithms for graph clustering.

Procedure. In Figure 2, we illustrate the steps of the point-to-point algorithm by employing the PPSP algorithm on an undirected graph. Figure 2(a) illustrates the structural details of the graph, with v_1 serving as the source vertex and v_9 serving as the destination vertex. The corresponding edge weights are denoted as $W_{(v_1,v_9)}$. The query is represented as $Q(v_1 \rightarrow v_9)$. In Figure 2(b), specific details related to the query are tracked, including active vertices (v_i) and their corresponding query results ($R(v_1, v_i)$) from v_1 to v_i in each iteration. The query result signifies the shortest distance between the corresponding vertices, with its value dynamically updated during the iterations, and its interpretation varies with specific algorithms. Specifically, the best-known query result in the current iteration from the source vertex to the destination vertex ($R(v_1, v_9)$) is referred to as the **bound**. Initially, v_1 is the only active vertex, setting $R(v_1, v_1)$ to 0, and the query results for other vertices are initialized to infinity since they are unknown. Subsequent iterations involve traversing outgoing neighbors (v_j) of active vertices (v_i). If $R(v_1, v_i) + W_{(v_i,v_j)} < R(v_1, v_j) < bound$ or $R(v_1, v_i) + W_{(v_i,v_j)} < R(v_1, v_j) = bound = \infty$, then $R(v_1, v_j)$ is updated to the value of $R(v_1, v_i) + W_{(v_i,v_j)}$, and v_j becomes the active vertex in the next iteration. Otherwise, the corresponding path is pruned. In the first iteration, we traverse the outgoing neighbors (v_2 and v_6) of the starting point (v_1) and obtain their query results from the starting point ($R(v_1, v_2)$ and $R(v_1, v_6)$), which are equal to the weights of their edges with the starting point. The activated v_2 and v_6 are added to the set of active vertices (V_{active}) for the next iteration. In the second iteration, the algorithm identifies a valid path from the source to the destination vertex: $v_1 \rightarrow v_6 \rightarrow v_9$. The query result of this path ($R(v_1, v_9)$) is used as the bound. All paths with



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
Initial	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Iter-1	0	1	∞	∞	∞	2	∞	∞	∞	∞	∞	∞
Iter-2	0	1	Prune 5	∞	2	2	Prune 7	∞	Bound 5	4	Prune 5	∞
Iter-3	0	1	5	∞	2	2	7	∞	5	3	5	∞
Iter-4	0	1	5	∞	2	2	7	∞	Query Result 4	3	5	∞
Early Termination												
Iter-5	0	1	5	6	2	2	7	∞	4	3	5	∞
Iter-6	0	1	5	6	2	2	7	8	4	3	5	∞
Iter-7	0	1	5	6	2	2	7	8	4	3	5	11
Normal Termination												

(b)

Fig. 2. Point-to-point shortest path query $Q(v_1 \rightarrow v_9)$

query results worse than the bound are terminated early, leading to the pruning of paths to v_3, v_7, and v_{11}. As iterations progress, the bound value is continuously updated. In the fourth iteration, a better query path is discovered: $v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_{10} \rightarrow v_9$, and the bound is updated with $R(v_1, v_9)$. At this point, all outgoing edges of active vertices have been pruned, resulting in no active vertices in the next iteration. Consequently, the query terminates early, and the bound value at this point becomes the final query result. The described pruning steps effectively reduce redundant data access and computations in point-to-point queries.

Pruning Mechanism. The primary optimization for these algorithms lies in the pruning mechanism [5], which eliminates unnecessary path traversals, allowing iterative queries to converge more quickly. Taking PPSP as an example, this process can be summarized in three stages: 1) Establishing a valid query result as the initial bound; 2) Evaluate paths based on specific criteria. If a path's query result is better than the current bound, update the bound and continue exploration. Otherwise, prune the path to avoid further unnecessary computation; 3) The process continues until no paths' query results are better than the current bound. This signifies reaching the optimal solution, with the bound representing the final converged result. three-stage pruning processes exist for other point-to-point queries, though the specific details and comparison rules vary. Table 1 illustrates the diverse meanings of bounds and their corresponding comparison rules across different algorithms.

For the PPSP algorithm, the bound represents the distance value of the shortest path from vertex v_i to v_j in each iteration. The algorithm's rule compares the distance of different paths. Any paths with distances greater than the bound are pruned. In the case of the PPWP/PPNP algorithm, the bound signifies the maximum (minimum) edge weight among all reachable paths from the source vertex (v_s) to the destination vertex (v_d) during an iteration. The rule dictates that if any edge weight in the path is narrower (wider) than the current bound, the corresponding path is pruned. For the point-to-point version of the BFS algorithm, the bound denotes the minimum layer count from vertex v_s to v_d. In connectivity or reachability algorithms, the bound indicates the connectivity from vertex v_s to v_d in undirected and directed graphs, respectively. The three algorithms mentioned later have a unique characteristic: once they encounter any valid path, it signifies the discovery of the desired optimal path, and the query concludes at that point. For instance, during the first traversal, BFS determines the shortest path to the endpoint. Once the path's accessibility is confirmed for the first time, there is no need to search for other reachable paths.

Algorithm	Meaing of Bound	Criterion for Comparison
PPSP	shortest distance	disatance
PPWP	weight of widest edge	weight
PPNP	weight of narrowest edge	weight
BFS	shortest layer	number of layers
Reachability	undirected graph connectivity	reachability
Connectivity	directed graph connectivity	reachability

TABLE 1: Details of different point to point query algorithms

2.2 Performance Bottlenecks in Concurrent Point-to-Point Queries

While extensive researches have been conducted on optimizing single point-to-point queries, effectively handling concurrent queries remains a challenge. To systematically evaluate the performance challenges of concurrent point-to-point queries, we examine the LLC miss ratio and redundant computation ratio of different systems as the concurrency level varies within a batch. In this context, we consider the computation of path segments that overlap among different queries as a redundancy. We implemented the concurrent version of point-to-point query systems: PnP and SGraph, and extended the general concurrent processing system Glign with pruning capabilities to support point-to-point queries. Experiments were conducted using a total of 512 queries with varying concurrency levels on the TW dataset. Table 2 illustrates an increase in LLC miss rates as the concurrency level rises. Notably, Glign demonstrates robust support for concurrent tasks, achieving the lowest miss ratio under conditions. Additionally, our analysis of traversal results within query batches reveals a positive correlation between concurrency and the occurrence of redundant computation in all systems. This correlation arises because, as the number of concurrent queries increases, the proportion of path segments that are computed more than once also increases. Glign exhibits a higher redundancy computation ratio than other solutions due to its lack of point-to-point query optimizations. As a result, many paths that should be pruned are computed. In a task-driven execution module, threads operate independently, leading to increased contention for shared resources. This results in redundant data access and computation overhead, which causes performance bottlenecks for concurrent point-to-point queries.

	Number of concurrent queries	PnP	SGraph	Glign
LLC miss ratio	1	41.1%	35.9%	28.4%
	32	59.4%	54.0%	32.2%
	64	71.4%	67.2%	38.5%
	128	80.3%	76.7%	43.4%
	512	89.6%	84.7%	49.4%
Redundant computation ratio	1	-	-	-
	32	65.6%	51.9%	72.9%
	64	74.4%	64.9%	84.8%
	128	85.6%	72.1%	92.6%
	512	96.8%	80.4%	97.4%

TABLE 2: Performance analysis of systems with varying concurrency (512 PPSP queries on TW [10])

2.3 Our Motivation

The mentioned challenges hinder the effectiveness of current solutions for handling concurrent point-to-point queries, yet they also create an opportunity through the sharing of data access and computation. In real-world graphs, a small fraction of vertices is connected to the majority of edges, resulting in a characteristic power-law distribution. *We refer to these heavily interconnected vertices as* ***hot vertices*** *and term the convergent path obtained from executing point-to-point queries for pairs of hot vertices as* ***hot paths***. Through a thorough analysis of the concurrent query process, we observe that hot vertices and hot paths account for the majority of redundant overhead. Using the PPSP example in Figure 2, queries $Q(v_1 \rightarrow v_9)$, $Q(v_1 \rightarrow v_{10})$, and $Q(v_2 \rightarrow v_9)$ independently recomputed the path from v_2 to v_{10}, incurring redundant overhead. For these queries, v_2, v_5, and v_{10} are identified as hot vertices, and the path $v_2 \rightarrow v_5 \rightarrow v_{10}$ is termed a hot path. Therefore, by identifying hot vertices and sharing information about hot paths, we can significantly reduce redundant computations and enhance the efficiency of concurrent point-to-point queries. Further details are provided below.

Data Access Sharing. Queries originating from different points often access the same vertices during traversal. In Figure 3, we record the traversal paths of concurrent queries within the same batch and analyze the proportion of vertices accessed more than once by different queries to the total activated vertices. When the query concurrency in a batch reaches 512, over 90% of the vertices are accessed more than once. However, in the existing task-driven execution module, different queries independently access their required data, resulting in redundant access to the same data. *Enabling queries to share access to the same graph structure data can amortize the data access overhead, significantly improving the overall throughput of concurrent query processing.*

Computation Sharing. As depicted in Figure 1, various queries frequently recompute the results of identical hot path segments during traversal, resulting in redundant computations. In Figure 4, we identify the intersection of various query paths, redundantly processed by all queries in a batch. We then select the top 522 hot vertices, constituting less than 1% of the total, and analyze their proportion in query intersections under different concurrency levels. As concurrency increases, the total amount of overlapping data decreases, while the percentage of hot vertices in this overlap can increase to 95% (at 512 concurrency). This indicates that hot vertices and their corresponding hot paths

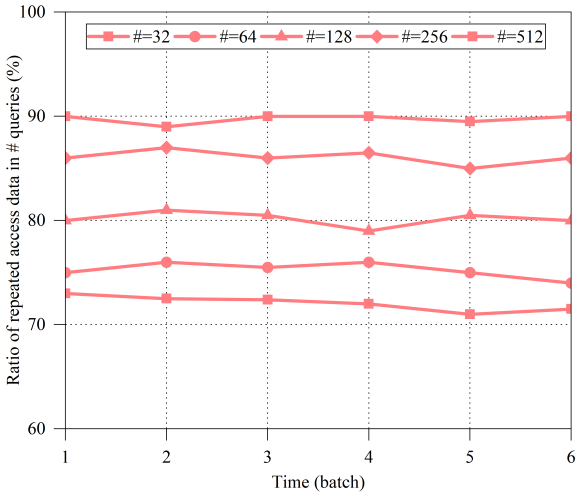


Fig. 3. Data access redundancy among concurrent queries

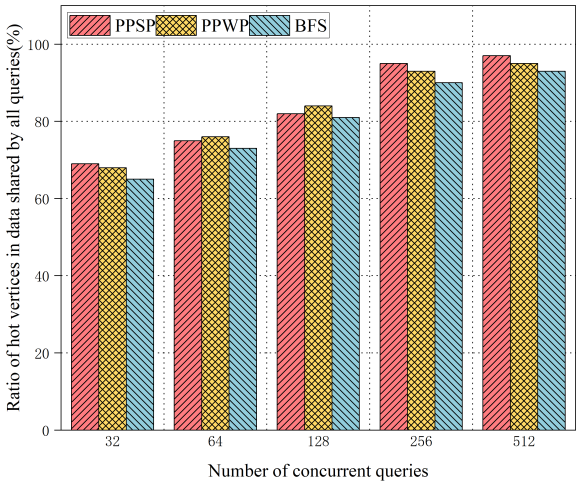


Fig. 4. Computation redundancy among concurrent queries

significantly impact redundant computations. *Facilitating the sharing of computations for hot paths among distinct queries can significantly mitigate computational overhead and enhance query efficiency.*

3 Overview Of GraphCPP

To enhance the efficiency of concurrent point-to-point queries, we introduce GraphCPP, a system designed to leverage data access and computation sharing among concurrent queries. It incorporates a novel data-driven caching execution mechanism to amortize data access overhead among multiple tasks. Additionally, it introduces a dual-level computation sharing mechanism, enabling computations for hot vertices and hot paths to be shared across queries. In this section, we will present an overview of GraphCPP. Section 3.1 introduces the overall architecture. Section 3.2 describes the data access sharing mechanism, and Section 3.3 describes the computation sharing mechanism.

3.1 System Architecture

GraphCPP adopts a four-layer architecture, as depicted in Figure 5. The **storage layer** stores the graph structure

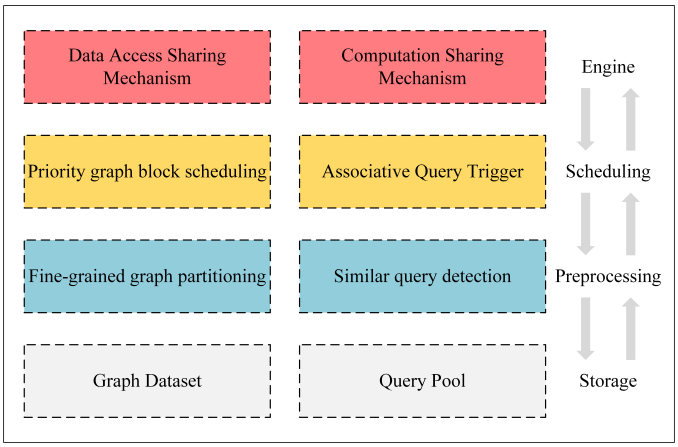


Fig. 5. System architecture diagram

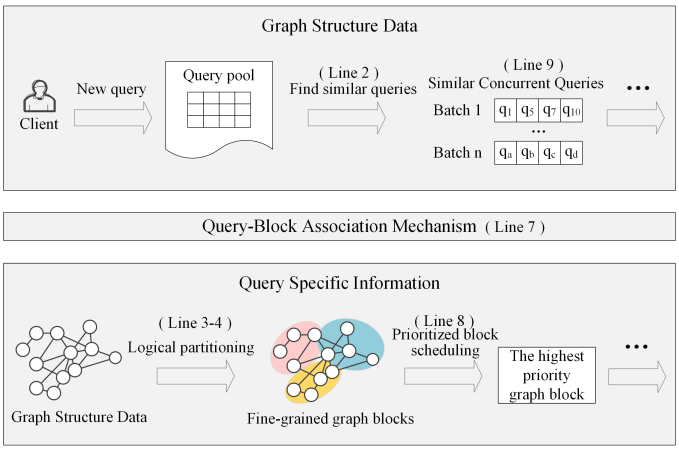


Fig. 6. Data Access Sharing Mechanism

data and the query specific information. In the **preprocessing layer**, graph structure information in memory is logically divided into fine-grained graph blocks, which can be suitably accommodated within LLC. Additionally, a query similarity-aware mechanism is implemented to cluster queries from the query pool into groups based on their similarities, thereby enhancing data access sharing efficiency by concurrently executing queries in a batch. The **scheduling layer** conducts scheduling based on the associations between blocks and queries. In this context, the term 'association' indicates that a specific graph block contains active vertices required by a particular query. This layer will establish associations between blocks and queries, as well as quantify the number of tasks associated with each graph block. It consists of two main components: 1) Priority Block Scheduling: Blocks with a higher number of associated tasks are prioritized for scheduling because they benefit more from sharing. 2) Associated Task Trigger: Once the scheduling layer identifies blocks for shared execution, it proceeds to determine all queries associated with these blocks. These associated queries form a subset of the query set. Simultaneously processing these associated tasks on shared graph blocks enables multiple queries to leverage the same underlying graph structure data, effectively reducing the data access overhead. The **engine layer** is the core of GraphCPP, and it consists of two components: 1) Data Access Sharing Mechanism: This mechanism identifies shared graph structure data accessed by multiple tasks, and employs a data-driven scheduling approach to enable concurrent queries to share the same data. 2) Dual-level Computation Sharing Mechanism: This mechanism exploits the fact that different queries often compute the results for the same hot paths. By facilitating the pre-computing and sharing of these computations across different queries, this mechanism optimizes query processing and effectively reduces computation redundancy.

3.2 Data Access Sharing Mechanism

In Section 2, we observed a significant overlap in graph structure data accessed by concurrent queries. However, under the current processing approach, this overlap data cannot be utilized effectively. For monotonic point-to-point queries on the graph, the order of data access does not affect the accuracy of the final results. Thus the core idea

Algorithm 1 Data Access Sharing Algorithm

```
1:  function DataAccessSharing(Q, G)
2:      Q_similar ← FindQueries(Q)
3:      S_block ← DetermineSharedBlockSize()
4:      T_block ← LogicalPartition(G, S_block)
5:      V_active ← InitializeActiveVertexSet(Q_similar)
6:      while V_active is not empty do
7:          UpdateAssociationInfo(V_active, T_block)
8:          b_i ← BlockPrioritizeSchedule()
9:          Q_{b_i} ← GetAssociatedQueries(b_i)
10:         V_active ← ComputationSharing(b_i, Q_{b_i})
11:     end while
12:     return R
13: end function
```

of our data sharing mechanism is to transform the conservative sequential, coarse-grained task-driven scheduling order into a concurrent, fine-grained data-driven approach. This approach addresses data redundancy among concurrent queries to amortize data access overhead, improve cache utilization efficiency, and consequently, enhance system throughput. The pseudo-code in Algorithm 1 illustrates the implementation of the Data Sharing Mechanism, which is consistent with the step in Figure 6.

The function, named `DataAccessSharing`, accepts two parameters: Q and G, representing all requests in the query pool and the graph structure data, respectively (line 1). The algorithm commences by identifying query set within the query pool, denoted as $Q_{similar}$ (line 2). Subsequently, the size of shared data blocks (S_{block}) is determined (line 3). We then employ a logical partitioning approach to subdivide coarse-grained graph partitions into fine-grained blocks (S_{block}). The information of logical partitioning is stored in T_{block}, a dictionary data structure that records the mapping relationship between physical data and logical partitions (line 4). Following this, the active vertex set (V_{active}) is initialized based on the $Q_{similar}$ (line 5). The algorithm iterates until V_{active} is empty (line 6). In each iteration, the association information between active vertices and graph blocks is updated (line 7). The algorithm prioritizes the selection of graph blocks using the `BlockPrioritizeSchedule` function

(line 8). Then, it retrieves the associated tasks using the `GetAssociatedQueris` function (line 9). Next, computation sharing is performed on the selected block and query set (line 10). This updates V_{active} for the next iteration. When all active vertices have been processed, indicating that the query batch has been completed, the algorithm returns the computation results (line 12). This part provides an overview of the algorithm's flow, with further details to be explored in the following parts.

Find Similar Queries. The query pool often contains numerous queries that exceed the concurrent limit. Hence, it's necessary to group these queries for batch processing. Queries within the same batch run concurrently, while those in different batches are processed sequentially. The degree of overlap among query paths can vary significantly, with some sharing many common path segments and others having minimal overlap. Therefore, the selection of query tasks within a batch can significantly impact overall system performance. To address this, we employ an intuitive similarity-aware approach for grouping queries within a batch. It calculates the similarity between queries q_i and q_j using the following formula, where v^i_{src} and v^i_{dst} denote the source and destination vertices of query q_i respectively, and $D(v_i, v_j)$ represents the distance between two vertices (v_i and v_j) using PPSP. The corresponding value increases as the similarity between queries rises. Specifically, we randomly select a query q_i and compute its similarity with other queries. If the similarity between a task and q_i exceeds a predefined threshold (we set it to 0.1 in our experiments), it's considered a similar query to q_i. Once enough similar queries fill a batch, they are executed concurrently. Since similarity computation can be performed in the background while executing query tasks, the overall overhead is minimal.

$$Similarity = \frac{1}{D(v^i_{src}, v^j_{src}) + D(v^i_{dst}, v^j_{dst})} \tag{1}$$

Determine shared block size. Cache plays a critical role in hierarchical storage systems. Therefore, it is important to select an appropriate shared graph block size to maximize cache utilization. Ideally, the shared graph block size should be small enough to fit into the LLC, yet not so small as to cause excessive scheduling overhead during query processing. Formula 2 is employed to calculate the maximum size for shared graph blocks, taking into account factors including shared block size (S_{block}), partition size (S_{graph}), total number of vertices ($|V|$), average storage cost per vertex (S_{vertex}), number of concurrent queries (N), LLC size (S_{LLC}), and reserved redundant space ($S_{reserve}$) determined by empirical judgment. By considering both graph structure and query-specific information, this formula enables us to determine the maximum granularity for each shared graph block while ensuring efficient utilization of the available LLC capacity.

$$S_{block} + \frac{S_{block}}{S_{graph}} \cdot |V| \cdot S_{vertex} \cdot N \leq S_{LLC} - S_{reserve} \tag{2}$$

Logical partitioning. After determining the shared block size, we apply this granularity to perform logical graph partitioning. Given that each edge in the graph structure dataset occupies a fixed size, we can calculate the maximum number of edges a shared block can accommodate. Subsequently, we partition the edges in the edgelist based on this calculated value. Since the edgelist is arranged in ascending order by source vertex ID, it suffices to record the start and end vertex IDs for each block. Due to variations in the out-degree of each vertex, resulting in a uniform division may amortize a vertex's different out-edges across multiple blocks. Therefore, we additionally store the count of edges associated with a specific vertex in a given block. This approach allows us to partition the physically contiguous graph structure data into logically independent blocks. The partitioning information is stored in T_{block}, and during data access, we can determine the physical location of a block by referring to the vertex ID and edge count saved in T_{block}.

Update the association information between blocks and queries. In our data-driven scheduling model, it is essential to update the association between blocks and queries in each iteration. The dynamic relationship between a graph block and a query is determined by the query's active vertex set (V_{active}). If a graph block contains active vertices required by a query, then the block is associated with the query. By tracking each query's traversal process during an iteration, we can predict V_{active} for the next round. Leveraging the T_{block}, we efficiently map these active vertices to their corresponding graph blocks, updating the graph block-query association. Although this association dynamically updates with each execution, the low number of active vertices ensures minimal overhead for maintaining these connections.

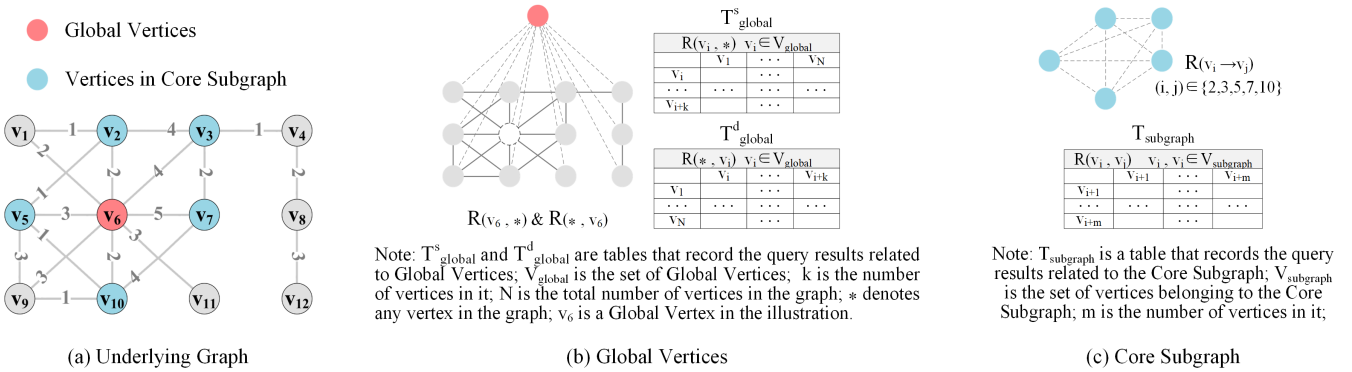
Block prioritize schedule. Following the established associative relationship between blocks and queries, we implement prioritized scheduling for graph blocks to address data access redundancy among concurrent queries. This approach prioritizes blocks with the greatest potential for shared data utilization across tasks, ultimately enhancing system performance. The calculation of priority follows the formula below.

$$Priority = w_1 \cdot N_{query} + w_2 \cdot \left(\frac{N_{hot}}{N_{active}}\right) \tag{3}$$

Here, N_{query} denotes the number of queries associated with the block, and $\frac{N_{hot}}{N_{active}}$ represents the proportion of active hot vertices within a block out of the total active vertices. w_1 and w_2 are weight parameters that balance the influence of these factors, with their values manually tuned. In our experiments, w_1 is set to 0.5, and w_2 is set to 1. The computed priority determines the sequence for loading graph blocks into the LLC, giving preference to those with higher priority and thus facilitating efficient data sharing.

3.3 Computation Sharing Mechanism

Recognizing that most of this redundancy is attributed to hot vertices and hot paths in Section 2.3, we introduce a dual-level computation sharing mechanism to share their query results among different queries. Query results represent the converged outcomes of executing the respective point-to-point query algorithm for a specific vertex pair. The operational process of this mechanism is illustrated in Figure 7:



Note: T^s_{global} and T^d_{global} are tables that record the query results related to Global Vertices; V_{global} is the set of Global Vertices; k is the number of vertices in it; N is the total number of vertices in the graph; $*$ denotes any vertex in the graph; v_6 is a Global Vertex in the illustration.

Note: $T_{subgraph}$ is a table that records the query results related to the Core Subgraph; $V_{subgraph}$ is the set of vertices belonging to the Core Subgraph; m is the number of vertices in it;

(a) Underlying Graph (b) Global Vertices (c) Core Subgraph

Fig. 7. Computation Sharing Mechanism

In Figure 7(a), we label vertices with connection edges exceeding 6 as **hot vertices**, considering the one with the highest degree as the Global Vertex, while the remaining hot vertices are allocated to the Core Subgraph. Figure 7(b) depicts the operation of the **Global Vertices Mechanism**, which computes query results from hot vertices to other vertices ($Q(v_6 \rightarrow *)$) and from other vertices to hot vertices ($Q(* \rightarrow v_6)$) during the preprocessing stage. This precomputation enables rapid determination of query bounds without actual computations when the query process traverses Global Vertices. For instance, before traversing the query $Q(v_1 \rightarrow v_9)$, we can quickly establish a valid path connecting v_1 and v_9: $v_1 \rightarrow v_6 \rightarrow v_9$. At this point, $Q(v_1 \rightarrow v_9) = R(v_1 \rightarrow v_6) + R(v_6 \rightarrow v_9)$. The right side of the equation represents the query results for v_1 to v_6 and v_6 to v_9, which are computed during the preprocessing stage. Using this equation, we obtain the bound for $Q(v_1 \rightarrow v_9)$ without redundant computation, assisting subsequent iterative processes. The computation sharing based on Global Vertices is termed as the first level of computation sharing. While the number of Global Vertices is often limited in practical applications due to their computational and storage costs, it also affects the efficiency of computation sharing. The subsequent Core Subgraph Mechanism addresses this limitation.

The **Core Subgraph Mechanism** is illustrated in Figure 7(c). During the preprocessing stage, we identify hot vertices within the Core Subgraph and pair them to create vertex pairs for queries. For a given number of hot vertices, denoted as m (in this instance, m is 5), we generate $m \times m$ vertex pairs and utilize a table ($T_{subgraph}$) of this size to store their query results. We consider the converged query paths as edges in Core Subgraph, with the edge weights indicating the corresponding converged query results. To keep operations lightweight, we do not directly compute the query results for these hot vertex pairs. Instead, during runtime, we extract hot path fragments from known query results. This is facilitated by a straightforward observation: *if we acquire a converged query path between two vertices, any path fragment extracted from this path also satisfies the convergence condition*. In the final part of this section, we elaborate on obtaining relevant information from converged paths, extracting hot path segments, and efficiently facilitating the sharing of query results for identical hot paths across different queries. As shown in Figure 7, leveraging the

Algorithm 2 Dual-level Computation Sharing Algorithm

```
1: function PREPROCESSING(V, k, m)
2:     V_global, V_subgraph ← StatisticsVertexDegree(V, k, m)
3:     T^s_global, T^d_global ← ComputeGlobalVertices(V_global)
4:     T_subgraph ← InitCoreSubgraph(V_subgraph)
5: end function
6: function COMPUTATIONSHARING(B, Q_similar)
7:     for each q_i in Q_similar in parallel do
8:         bound ← L1Share(T_global, q_i)
9:         V_active ← GetActiveVertexSet(q_i)
10:        for each v_i in V_active do
11:            if v_i ∈ V_subgraph then
12:                bound ← L2Share(v_i, T_subgraph, bound)
13:            end if
14:            bound ← travNeighbors(v_i, V_active, bound)
15:        end for
16:    end for
17:    return V_active
18: end function
19: function MAINTAINCORESUBGRAPH(R)
20:    T_subgraph ← ExtractHotPath(R)
21:    return T_subgraph
22: end function
```

Core Subgraph allows us to break down $Q(v_1 \rightarrow v_9)$ into $Q(v_1 \rightarrow v_2) + R(v_2 \rightarrow v_{10}) + R(v_{10} \rightarrow v_9)$, effectively avoiding redundant computations of $Q(v_2 \rightarrow v_{10})$. It is crucial to note that during querying, we lack knowledge on how to partition a complete query into multiple subqueries. In practice, we treat edges in the Core Subgraph as normal outgoing edges. When a task traverses a vertex v_i within the Core Subgraph, it considers other hot vertices in the Core Subgraph as outgoing neighbors of v_i, with the edge weights representing the corresponding query results. Therefore, for $Q(v_1 \rightarrow v_9)$, v_1 initially traverses its outgoing neighbor v_2, then v_2 traverses its outgoing neighbor v_{10}, and finally, v_{10} traverses its outgoing neighbor v_9. The entire process aligns with the normal traversal steps, rendering the Core Subgraph transparent to the user. In contrast to the Global Vertices Mechanism, the Core Subgraph is more lightweight, encompassing more hot vertices and providing more precise pruning bounds.

Algorithm 2 illustrates the steps of the dual-level com-

putation sharing. It is broadly divided into three phases: the preprocessing phase (lines 1-5), which is responsible for collecting vertex degrees, precomputing Global Vertices, and initializing the Core Subgraph. The computation sharing phase (lines 6-18) achieves the first level of computation sharing through Global Vertices mechanism (line 8) and the second level through the Core Subgraph (line 12). The phase of Maintaining the Core Subgraph (lines 19-22) involves extracting hot paths from existing computation results into the Core Subgraph, thereby reducing redundant computations for subsequent queries. The first stage is executed only once at the beginning of the query, while the latter two stages are iteratively executed throughout the entire runtime. Below are explanations of key functions in the algorithm.

Preprocessing. Instead of using a complex evaluation method [45], we opt for a simple yet effective approach by selecting a subset of vertices with the highest degree of incoming and outgoing edges in the graph as hot vertices. In the preprocessing stage, we utilize the selection sort algorithm to identify the top 522 vertices with the highest degree. We designate the top 10 vertices as Global Vertices and allocate the remaining 512 vertices to the Core Subgraph (line 2). For the Global Vertices mechanism, employ a point-to-all query algorithm to obtain all query results for paths originating from or leading to Global Vertices. This generates $k*|V|*2$ computation results, where k represents the number of Global Vertices, and $|V|$ represents the total number of vertices in the graph. We utilize T^s_{global} to store all query results with Global Vertices as source, and T^d_{global} to store those with Global Vertices as destination. The row and column indices of the table correspond to the starting and ending points of the query, facilitating the quick retrieval of the corresponding query results (line 3). For the Core Subgraph mechanism, it contains m vertices, allowing the formation of $m * m$ unique vertex pairs, with each pair corresponding to a single query result. We use a dedicated two-dimensional table $T_{subgraph}$ to store these results. Unlike the Global Vertices mechanism, the Core Subgraph only establishes the table and initializes it with empty values during preprocessing. In the third stage, it extracts hot paths from the previously computed query results to populate the Core Subgraph (line 4).

Computation Sharing. In the computation sharing mechanism, we execute all queries concurrently (line 7). **First layer of computation sharing**: when a query $Q(v_s \rightarrow v_d)$ starts, the `L1Share()` function traverses T^d_{global} to find all query results $R(v_s, *)$ corresponding to the row of v_s, and traverses T^s_{global} to find all query results $R(*, v_d)$ corresponding to the column of v_d. Then, it iterates through all Global Vertices v_h to obtain the minimum value of $R(v_s, v_h) + R(v_h, v_d)$. This value serves as the query's bound, which can be utilized for pruning (line 8). This bound is continuously updated in successive iterations. If the query's start and end points are unreachable, the bound is set to infinity. The system then retrieves query information to obtain the corresponding active vertex set (line 9). **Second layer of computation sharing**: as mentioned above, various queries frequently recalculate certain popular paths. The Core Subgraph is utilized to share the query results for these hot paths. When traversing the hot vertex v_h within the Core Subgraph, we initially explore $T_{subgraph}$ to retrieve all query results corresponding to the row of v_h. Each entry in this row represents a query result for a pair of vertices, where the row index denotes the starting vertex and the column index denotes the destination vertex. If there are non-empty entries in this row, it indicates that before the current query execution, another task traversed the corresponding path segment, obtaining and storing the relevant query result. This enables us to retrieve all non-empty query results from v_h to other hot vertices. We consider these hot vertices as out-edge neighbors of v_h, and the query results between them as the edge weights. These edges serve as shortcuts, reducing redundant computations for hot paths and accelerating traversal (line 12). Then, the query iterates over the neighbors of each query task, updating the bound and the next round of active vertex set (line 14). Finally, the function returns the updated set (V_{active}) for the next round of computation (line 17).

Maintain Core Subgraph. The first two stages initialize and utilize the Core Subgraph, respectively. This phase refines the query results of Core Subgraph in $T_{subgraph}$. In monotonic point-to-point query algorithms, any path fragment of a convergent path does remain convergent itself. For instance, if we obtain the shortest path between two vertices, any fragment of that path corresponds to the shortest path between those vertices. After completing a batch of queries, we analyze the composition of the convergent paths corresponding to the queries. If a known optimal query path contains multiple hot vertices, it implies the presence of at least one convergent path fragment consisting of hot vertices at both ends. So, how do we obtain convergent path information? As the query path traverses a vertex v_i, we record the query result for the path $v_s \rightarrow v_i$ and the parent node of v_i on this path. This information is retained until a new and better path reactivates v_i, at which point all information is updated. In this way, when the final query converges, we obtain the query results and parent nodes for each relevant vertex. Then, starting from the destination vertex (v_d), we extract the query results from the starting vertex to v_d and its corresponding parent node along the convergent path. We then repeat the previous step with the obtained parent node until we obtain the complete convergent path from v_s to v_d. Once this extraction process is complete, we update the query results in $T_{subgraph}$ at their corresponding positions to facilitate sharing in the next batch of queries. It's essential to note that due to the pruning operations employed in point-to-point queries, effectively avoiding a significant portion of redundant calculations, the proportion of activated vertices is minimal. Consequently, the process of extracting hot path query results can be completed with extremely low time and space overhead.

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

Hardware Configuration. The experiments are conducted on an 8-node cluster, each machine equipped with 2 32-core lntel Xeon platinum 8358 CPUs (each CPU has 48MB L3 cache) and 256 GB memory. All nodes are interconnected through an lnfiniband network with a bandwidth of 300Gbps. The programs are compiled using gcc version 9.4.0, openMPl version 4.0.3, and with openMP enabled.

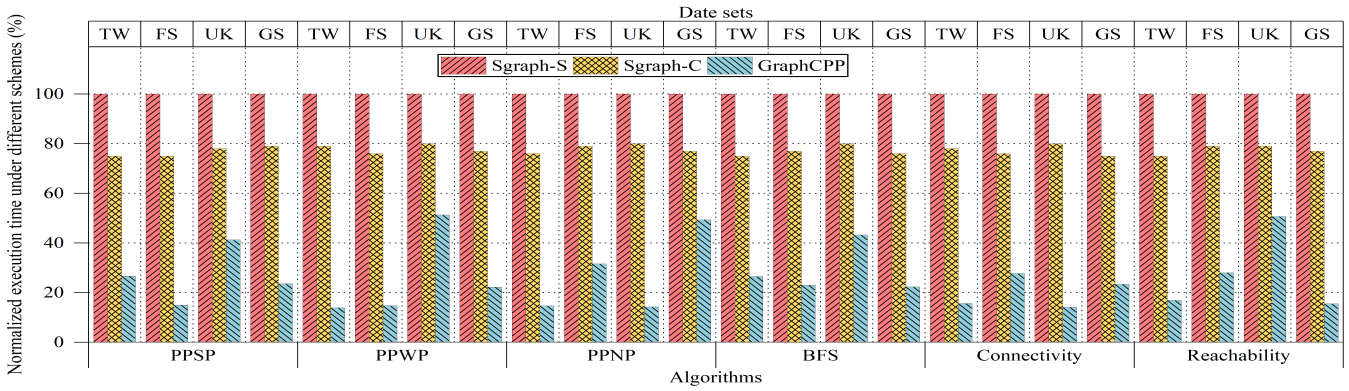


Fig. 8. Total execution time of different systems with six algorithms on four datasets (512 queries, 128 concurrent, normalized to SGraph-S)

Graph Algorithms And Datasets. We utilized six commonly used point-to-point query algorithms, as shown in Table 1. These algorithms have broad applications in different domains, such as social networks and traffic planning, making them representative. Additionally, Table 3 provides details on the graph datasets used in these algorithms. The datasets cover diverse application scenarios and scales, following a power-law distribution that accurately reflects the characteristics of real-world graph distributions. We adopt advanced graph partitioning approach [44] to meet the requirements of distributed systems.

Datasets	Vertices	Edges	Data sizes
Twitter-2010(TW) [10]	41.7M	1.5B	10.9GB
Friendster(FS) [9]	65.6M	1.81B	8.7GB
Gsh-2015-host(GS) [11]	68.7M	1.8B	13.4GB
UK-2007-05(UK) [12]	106M	3.74B	27.9GB

TABLE 3: Graph Dataset Information

System Comparison. We conducted a comparative analysis between GraphCPP and the state-of-the-art point-to-point query solution SGraph [5]. Additionally, we implemented a concurrent query processing version of SGraph, denoted by the '-C' suffix, and a sequential query execution version, denoted by the '-S' suffix. To ensure scientific rigor in our experiments, we used random parameters for each query, maintaining consistency with the baseline: the number of Global Vertices was set to 10, and the vertices of the Core Subgraph in GraphCPP were set to 512. All tests were conducted 10 times, and the reported results represent the average values.

4.2 Overall Performance Comparison

Figure 8 illustrates the overall execution time for 512 queries using various approaches. The execution time is normalized relative to the performance of SGraph-S, considering significant variations in the execution time across different test cases. The results indicate that GraphCPP consistently achieves shorter execution time for all graphs and algorithms. Specifically, when compared to alternative schemes, GraphCPP demonstrates an average throughput improvement ranging from 1.56 to 5.67 times. This improvement is attributed to the reduction of data access costs and effective pruning of the Core Subgraph in GraphCPP.

For a more detailed performance analysis, we further categorize the total time into data access time and graph processing time, measured by the waiting time for core pauses. As depicted in Figure 9, GraphCPP requires less time for graph data access compared to SGraph-S, with this proportion decreasing as the graph size increases. Notably, GraphCPP's data access time averages a reduction of 1.23 to 2.39 times when compared to other systems. The efficiency of GraphCPP stems from two critical factors: 1) identical portions of graph data required by different concurrent queries are loaded and maintained as a single copy in memory, reducing overall memory consumption; 2) graph data blocks are prioritized and regularly loaded into the LLC based on associated query counts, facilitating job reuse and effectively lowering LLC miss rates, thereby minimizing unnecessary memory data transfers. Furthermore, the two-level computing sharing mechanism contributes to GraphCPP's lower computation time compared to other systems.

4.3 Efficiency of Data Access Sharing Mechanism

In Figure 9, GraphCPP demonstrates a substantial reduction in data access time, ranging from 45% to 85%, when utilizing the data access sharing mechanism compared to SGraph. This performance improvement is attributed to concurrent tasks sharing the data access overhead, and the unified scheduling approach of the data access sharing mechanism enhances data locality for various tasks. To assess the effectiveness of our data-sharing mechanism, we conducted 512 queries on the different graph and normalized GraphCPP's LLC miss ratio relative to SGraph-S. As shown in Figure 10, GraphCPP consistently achieves a lower LLC miss ratio across all datasets and algorithms, averaging a 50% improvement.

4.4 Efficiency of the Computation Sharing Mechanism

We compared the performance of GraphCPP and SGraph-S (Figure 9), demonstrating that despite the inherent overhead of the computation sharing mechanism (including the cost of computing Global Vertices and maintaining the Core Subgraph), an overall computational acceleration is achieved. The specific costs associated with this mechanism are outlined below:

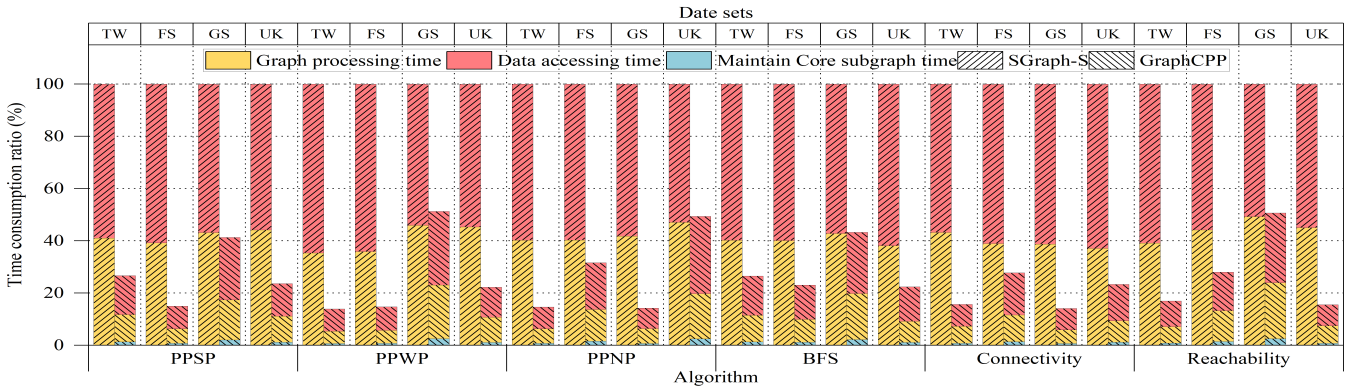


Fig. 9. Execution time breakdown of different systems with six algorithms on four datasets (512 queries, 128 concurrent, normalized to SGraph-S)

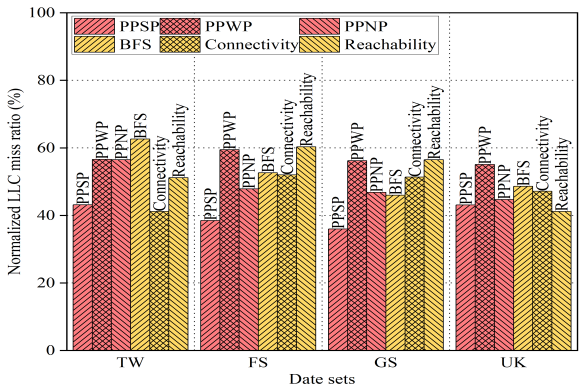


Fig. 10. LLC miss rate of GraphCPP with six algorithms on four datasets (512 queries, 128 concurrent, normalized to SGraph-S)

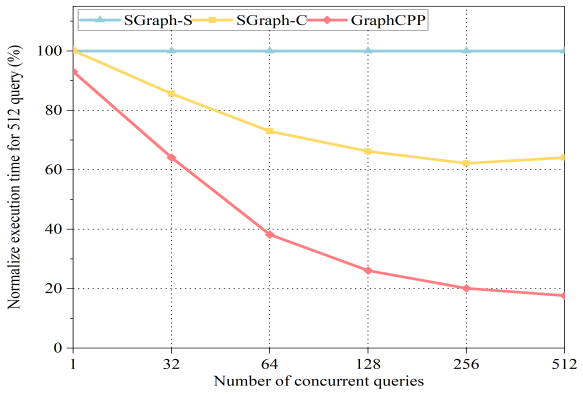


Fig. 11. Scalability of different systems with varying concurrency (512 PPSP queries on TW [10], normalized to SGraph-S)

Global Vertices Overhead. GraphCPP and SGraph-S both require precomputation and storage of queries involving Global Vertices in the graph, facilitating shared calculations for any computation that involves Global Vertices. The computation and storage overhead for Global Vertices is proportionate to the number of Global Vertices. For SGraph, it needs to maintain a sufficient number of Gobal Vertices (usually 16) to ensure the efficiency of computation sharing. However, for GraphCPP, it achieves this by initializing query boundaries based on Global Vertices, and more precise boundaries can be established in subsequent Core Subgraph computations. This flexibility enables GraphCPP to select fewer Global Vertices query results without compromising the efficiency of computation sharing. For instance, reducing the number of Global Vertices from 16 to 10 can result in a 37.5% reduction in the computation and storage overhead associated with Global Vertices.

Core Subgraph Overhead. We designed the construction of the Core Subgraph to leverage previous query results, employing a lightweight prefix sum method for extracting hot paths. As depicted in Figure 8, the time dedicated to maintaining the Core Subgraph constitutes approximately 5% of GraphCPP's total execution time. Despite this, it significantly reduces the overall computation time. Additionally, for each hot path, we need to store its query result and the vertices it traverses. We observe a positive correlation between the number of Core Subgraph vertices and storage overhead. Nevertheless, this cost remains relatively small when compared to the overall graph size since the Core Subgraph exclusively stores paths among hot vertices, and the maximum number of hot paths is merely the square of the hot vertices, making it quite limited compared to larger graphs.

4.5 Scalability of GraphCPP

In Figure 11, we evaluated the scalability of different systems by examining the time required for 512 queries under different concurrent query counts. For clearer comparison, we normalized the query time of SGraph-C and GraphCPP relative to SGraph-S. The blue horizontal line represents the time required for SGraph-S to linearly execute 512 queries. We chose the maximum of 512 concurrency, indicating that all queries can be completed in a single concurrent execution. SGraph-C benefits from concurrent execution; however, the absence of a data sharing mechanism results in the highest synchronization overhead among concurrent tasks, resulting in an earlier performance turning point. As the concurrent count increases, the performance gains diminish gradually. In comparison, GraphCPP consistently exhibits a decrease in processing time. This is attributed to its efficient data access and computation sharing mechanism, which consequently reduce the overhead of individual query task, enhancing the system's scalability to concurrent query tasks.

5 RELATED WORK

Graph Processing Systems. Graph computation systems fall into three main categories based on differences in storage media and computing platforms. 1) Single-node in-memory systems: Ligra [19] optimizes graph traversal using push-pull computation. GraphBolt [20] and ACGraph [21] handle streaming graphs incrementally. DiGraph [22] supports iterative directed graph processing on GPUs. DepGraph [23] and TDGraph [24] reduce redundancy and data access costs through topology-aware execution. 2) Single-node out-of-core systems: FlashGraph [25] achieves high IOPS with semi-external memory. GridGraph [26] enhances locality with Streaming-Apply. DGraph [27] accelerates processing through faster state propagation. GraphM [28] optimizes throughput for concurrent systems. 3) Distributed systems: Pregel [29] uses Bulk Synchronous Parallel (BSP) for scalable, fault-tolerant processing of large graphs. PowerGraph [34] reduces communication volume with vertex-cut partitioning. PowerLyra [36] employs a hybrid model for high/low-degree vertices. Gemini [31] ensures scalability with a dual-mode engine. CGraph [30] minimizes overhead with a load-trigger-push (LTP) model. However, existing graph computing systems mainly focus on point-to-all algorithms, which are not suitable for point-to-point scenarios.

Point-to-Point Queries. Numerous studies have explored point-to-point queries. For example, Hub^2 [37] introduces a specialized hardware accelerator that utilizes a hub-centric approach to constrain the search scope of high-degree vertices, thereby expediting the PPSP process. Quegel [38] pursues a software-based query result sharing approach, improving response speed by constructing a static distributed query-sharing table during loading. PnP [4] adopts a universal pruning strategy to reduce redundant accesses and computations. Tripoline [6] combines query reuse and pruning, achieving incremental query evaluation without prior knowledge through the application of the triangle inequality principle. Although initially designed for point-to-all algorithms, it can also be applied to point-to-point query systems. SGraph [5] further optimizes pruning strategies by incorporating upper and lower bounds, resulting in sub-second latency queries on large graphs with dynamically changing data. Existing solutions focus on improving the speed of individual point-to-point queries by pruning or sharing results. But they often overlook optimizing throughput in concurrent queries by sharing data access. Additionally, some approaches utilize the Global Vertices mechanism to achieve computation sharing. This approach requires exponential storage, computation, and dynamic upkeep as the graph size grows. To alleviate high overhead, practitioners often limit the number of vertices, which compromises the efficiency of computation sharing.

Concurrent Graph Computing. Concurrent graph processing systems address large-scale data across diverse architectures. Single-node in-memory system like Congra [39], enhances throughput and resource efficiency through dynamic query scheduling and awareness of atomic operations. Krill [40] adopts an SAP model to simplify attribute data management and reduce memory access. ForkGraph [41] accelerates execution by employing a yield-based scheduling strategy for efficient data sharing among concurrent queries. Out-of-core systems, such as GraphM [28] and its distributed counterpart CGraph [30], exploit locality to facilitate effective data sharing and computation. Other out-of-core systems, including Seraph [42], advocates for decoupling data structures, while MultiLyra [43] optimizes batch query evaluation by distributing communication costs through graph and boundary sharing. While the previously mentioned systems concentrate on optimizing data access sharing for concurrent general algorithms, they overlook specific optimizations for point-to-point queries.

6 CONCLUSION

Traditional point-to-point query systems excel in optimizing individual queries, but encounter throughput challenges in large-scale concurrent query scenarios. In contrast, contemporary concurrent graph computing systems have extensively explored the concurrent execution of point-to-all algorithms. However, due to the absence of efficient pruning mechanisms, they demonstrate low efficiency in handling point-to-point queries. GraphCPP identifies substantial redundancies in data access and computation among concurrent point-to-point queries, which brings optimization opportunities. It introduces a data-driven caching execution mechanism to facilitate overlapping data access sharing during concurrent queries. Additionally, it employs a two-level computation sharing mechanism, effectively enabling computation sharing across multiple queries. Experimental results indicate that GraphCPP achieves a performance improvement of 3.2x on average compared to existing state-of-the-art approach.

7 ACKNOWLEDGMENTS

This paper is supported by National Key Research and Development Program of China (No. 2022YFB2404202), Key Research and Development Program of Hubei Province (No. 2023BAB078), Knowledge Innovation Program of Wuhan-Basi Research (No. 2022013301015177), and Huawei Technologies Co., Ltd (No. YBN2021035018A6).

REFERENCES

[1] "Google Maps," https://www.alibabagroup, 2023.
[2] "Facebook," www.facebook.com, 2023.
[3] "Alipay," www.alipay.com, 2019.
[4] C. Xu, K. Vora, R. Gupta, "Pnp: Pruning and prediction for point-to-point iterative graph analytics," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, Providence, RI, USA, 2019, pp. 587–600.
[5] H. Chen, M. Zhang, K. Yang, et al., "Achieving Sub-second Pairwise Query over volving Graphs," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, Vancouver, BC, Canada, 2023, pp. 1–15.
[6] X. Jiang, C. Xu, X. Yin, et al., "Tripoline: generalized incremental graph processing via graph triangle inequality," in *Proc. Eur. Conf. Comput. Syst.*, United Kingdom, 2021, pp. 17–32.
[7] X. Yin, Z. Zhao, and R. Gupta, "Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing," in *Proc. ACM Int. Conf. Architectural Support Program. Languages Operating Syst.*, Vancouver, BC, Canada, 2022, pp. 78–92.
[8] "Cagis," http://www.cagis.org.cn, 2021.
[9] "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html, 2020.
[10] H. Kwak, C. Lee, H. Park, et al., "What is Twitter, a social network or a news media?" in *Proc. Int. Conf. World Wide Web*, Raleigh, NC, USA, 2020, pp. 591–600.

[11] P. Boldi, A. Marino and M. Santini, et al., "BUbiNG: Massive Crawling for the Masses," *ACM Trans. Web*, vol. 12, no. 2, pp. 1–26, 2018.
[12] P. Boldi, M. Santini and S. Vigna, "A large time-aware web graph," *ACM SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.
[13] K. Joseph, H. Jiang, "Content based News Recommendation via Shortest Entity Distance over Knowledge Graphs," in *Companion Proc. 2019 World Wide Web Conf.*, San Francisco, CA, USA, 2019, pp. 690–699.
[14] M. Pollack, "The maximum capacity through a network," *Operations Res.*, Vol. 8, no. 5, pp. 733–736, 1960.
[15] O. Berman, G. Handler, "Optimal Minimax Path of a Single Service Unit on a Network to Nonservice Destinations," *Transp. Sci.*, Vol. 21, no. 2, pp. 115–122, 1987.
[16] S. Schaeffer, "Graph clustering," *Comput. Sci. Rev.*, Vol. 1, no. 1, pp. 27–64, 2020.
[17] L. Dhulipala, C. Hong, J. Shun, "A framework for static and incremental parallel graph connectivity algorithms," *arXiv preprint arXiv:2008.03909*, 2020.
[18] E. Cohen, E. Halperin, H. Kaplan, et al., "Reachability and Distance Queries via 2-Hop Labels," *SIAM J. Comput.*, Vol. 32, no. 5, pp. 1338–1355, 2003.
[19] J. Shun, J. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. Symp. Princ. Pract. Parallel Program.*, Shenzhen, China, 2013, pp. 135–146.
[20] M. Mariappan, K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. EuroSys Conf.*, Dresden, Germany, 2019, pp. 1–16.
[21] Z. Jiang, et al., "ACGraph: Accelerating Streaming Graph Processing via Dependence Hierarchy," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2023.
[22] Y. Zhang, X. Liao, H. Jin, et al., "DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs," in *Proc. Int. Cinf. Architectural Support Program. Languages and Operating Syst.*, Providence, RI, USA, 2019, pp. 601–614.
[23] Y. Zhang, X. Liao, H. Jin, et al., "DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing," in *IEEE Int. Symp. High-Perform. Comput. Architecture* , Seoul, Korea, 2021, pp. 371–384.
[24] J. Zhao, Y. Yang, Y. Zhang, et al., "TDGraph: a topology-driven accelerator for high-performance streaming graph processing," in *Proc. Annu. Int. Symp. Comput. Architecture*, New York, NY, USA, 2022, pp. 116–129.
[25] D. Zheng, M. Mhembere, R. Burns, et al., "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs," in *USENIX Conf. File Storage Technologies*, Santa Clara, CA, USA, 2015, pp. 45–58.
[26] X. Zhu, W. Han, W. Chen, "GridGraph:Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning," in *USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, 2015, pp. 375–386.
[27] Y. Zhang, X. Liao, X. Shi, et al., "Efficient Disk-Based Directed Graph Processing: A Strongly Connected Component Approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 830–842, 2018.
[28] J. Zhao, Y. Zhang, X. Liao, et al., "GraphM: an efficient storage system for high throughput of concurrent graph processing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage and Anal.*, Denver, Colorado, USA, 2019, pp. 1–14.
[29] G. Malewicz, M. Austern, A. Bik, et al., "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2010, pp. 135–146.
[30] Y. Zhang, J. Zhao, X. Liao, et al., "CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing," in *USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2018, pp. 441–452.
[31] X. Zhu, W. Chen, W. Zheng, et al., "Gemini: A Computation-Centric Distributed Graph Processing System," in *USENIX Symp. Operating Syst. Des. Implementation*, Savannah, GA, USA, 2016, pp. 301–316.
[32] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Proc. Hadoop Summit*, pp. 5–9, 2011.
[33] Y. Low, J. Gonzalez, A. Kyrola, et al., "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2019.
[34] J. Gonzalez, Y. Low, H. Gu, et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *USENIX Symp. Operating Syst. Des. Implementation*, Hollywood, CA, USA, 2012, pp. 17–30.
[35] J. Gonzalez, R. Xin, A. Dave, et al., "GraphX: Graph Processing in a Distributed Dataflow Framework," in *USENIX Symp. Operating Syst. Des. Implementation*, Broomfield, CO, USA, 2014, pp. 599–613.
[36] R. Chen, J. Shi, Y. Chen, et al., "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, no. 3, pp. 1–39, 2019.
[37] R. Jin, N. Ruan, B. You, et al., "Hub-accelerator: Fast and exact shortest path computation in large social networks," *arXiv preprint arXiv:1305.0507*, 2013.
[38] Q. Zhang, D. Yan and J. Cheng, "Quegel: A general-purpose system for querying big graphs," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2016, pp. 2189–2192.
[39] P. Pan and C. Li, "Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines," in *IEEE Int. Conf. Comput. Des.*, Boston, MA, USA, 2017, pp. 217–224.
[40] H. Chen, M. Shen, N. Xiao, et al., "Krill: a compiler and runtime system for concurrent graph processing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage and Anal.*, New York, NY, USA, 2021, pp. 1–16.
[41] S. Lu, S. Sun, J. Paul, et al., "Cache-efficient fork-processing patterns on large graphs," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2021, pp. 1208–1221.
[42] J. Xue, Z. Yang, Z. Qu, et al., "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proc. Int. Symp. High-perform. Parallel Distrib. Comput.*, Vancouver, BC, Canada, 2014, pp. 227–238.
[43] A. Mazloumi, X. Jiang, R. Gupta, "Multilyra: Scalable distributed evaluation of batches of iterative graph queries," in *IEEE Int. Conf. Big Data*, Los Angeles, CA, USA, 2019, pp. 349–358.
[44] X. Liu, Z. Ji, T. Hou, "Graph partitions and the controllability of directed signed networks," in *Sci. China Inf. Sci*, vol. 62, no. 4, pp. 1–11, 2019.
[45] J. Zhang, H. Chen, D. Yu, et al., "Cluster-preserving sampling algorithm for large-scale graphs," in *Sci. China Inf. Sci*, vol. 66, no. 1, pp. 1–17, 2023.