目录

[动态图上数据驱动的并发点对点查询系统 2](#_Toc149671637)

[摘要 2](#_Toc149671638)

[前言 3](#_Toc149671639)

[背景和动机 6](#_Toc149671640)

[Preliminaries 7](#_Toc149671641)

[并发点对点查询任务的性能瓶颈 8](#_Toc149671642)

[我们的启发 10](#_Toc149671643)

[系统概述 11](#_Toc149671644)

[系统架构 11](#_Toc149671645)

[整体执行流程 12](#_Toc149671647)

[数据访问共享机制 12](#_Toc149671648)

[计算共享机制 17](#_Toc149671649)

[其它优化 20](#_Toc149671651)

[实验评估 21](#_Toc149671652)

[相关工作 22](#_Toc149671653)

[结论 23](#_Toc149671654)

[废弃材料 25](#_Toc149671655)

[素材库： 25](#_Toc149671656)

某些变量用符号表示

待办：完善实验统计，补足图和数据部分。

动态图上数据驱动的并发点对点查询系统

摘要

随着图处理技术在地图导航、网络分析等领域的大范围应用，大量点对点查询任务在同一个底层图上并发运行，对图查询系统的吞吐量提出了极高的要求。然而已有的图查询系统聚焦于优化单次点对点查询的速度，在处理并发图计算时，由于冗余的数据访问开销和计算开销，整体吞吐量很差。

我们观察到由于图数据存在幂律分布的特点，不同的查询的遍历路径往往在少量高度顶点组成的局部路径上彼此重叠，体现出了并发点对点查询任务的数据相似性，这启发我们提出一个数据驱动的并发点对点查询系统-GraphCPP。它采用了一个“数据驱动的缓存执行机制”，通过细粒度的图分块调度，实现了并发任务之间的**数据共享**，提高了数据访问效率；同时它通过“核心子图机制”对图中高度顶点之间的索引值进行预计算，以便在查询到来时快速确定高频共享路径段的索引值，实现了**计算共享**，加快查询结果的收敛。此外，我们通过“查询路径相似性预测机制”，在调度时从任务池中选择相似任务批量执行，更好地利用了并发点对点查询任务的数据相似性。我们将GraphCPP与最先进的点对点查询系统进行对比，包括SGraph[x]、Tripoline[x]、Pnp[x]，实验表明，GraphCPP将并发点对点查询的效率提升了xxxx倍。

索引关键词: graph process, point-to-point graph queries, concurrent jobs, data locality

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

Abstract

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

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

Index term: graph process, point-to-point graph queries, concurrent jobs, data locality

前言

图上的点对点查询任务指利用图这一通用数据结构，发掘两个特定对象之间的某种联系。和传统的图查询方法不同，图上的点对点查询专门针对两个特定顶点间的关联或路径进行分析，而无需关心整个图或其大规模子集的复杂查询。这种有针对性的查询策略赋予了点对点查询巨大的优化潜力。对于一些点对点查询版本的单调图查询算法，如Point-to-Point Best Path for SSSP(PPSP)、Point-to-Point Widest Path for SSWP(PPWP) 以及 Point-to-Point Narrowest Path for SSNP(PPNP)，可以在无需或少量查询与处理不相关的其他顶点或边的情况下，精确地确定两顶点之间的特定路径属性。由于点对点查询在图分析中的这种高效性，它在多个领域中都已得到广泛的实践应用。如：在物流运输时，找到两个地点之间的最佳路径；在社交网络分析时，通过查找两个用户之间的关系链，为用户推荐可能的朋友；在金融风险分析时，分析风险是如何从一个实体传播到另一个实体；这些热门应用提出了在同一个底层图上执行大规模并发点对点查询的需求。

然而已有的点对点查询的解决方案都聚焦于加速单次查询的效率，而忽略了对并发查询的优化。而要实现并发点对点查询，需要解决以下两个难题。

第一，实现数据共享。不同查询任务的遍历路径存在着大量重叠，然而在现有的执行体制下，并发任务之间的数据彼此隔离，不能共享重叠数据，从而造成了冗余数据访问。此外，不同任务对于相同的图结构数据的访问顺序不同，它们之间的步调不一致也增加了数据共享的难度。

第二，实现计算共享。图数据往往遵循幂律分布，少量高度顶点组成的路径段会频繁出现在不同查询的最佳路径中。由于高度顶点存在大量邻居顶点，不同任务对于它们的重复遍历常常导致计算开销的爆炸式增长。一些已有的系统中尝试采用全局索引的方式来进行计算共享，带来了昂贵的计算、存储、更新开销，这限制了计算共享的覆盖率和精确度。

INTRODUCTION

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

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

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

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

针对上述问题，我们设计了GraphCPP：一种数据驱动的并发点对点查询系统。

首先，它针对并发任务数据共享难题，提出了一种数据驱动的缓存执行机制，将传统的“任务→数据”的调度方式改为“数据→任务”的调度方式，进而实现多任务之间重叠图结构数据访问的共享。在这种执行机制下，GraphCPP会首先确定要数据的调度顺序，它将图结构数据从逻辑上划分为LLC级别的细粒度分块。接着根据查询任务活跃顶点集所处的图分块，将查询任务与其相关的图分块联系起来，任务的活跃顶点每轮都会变化，共享分块的关联任务数也需要每轮更新。统计分块的关联任务信息，关联任务数量越多的分块优先级越高，更可能被优先调度；为了实现“数据→任务”的调度方式，GraphCPP采用了一种关联任务触发机制。它按照优先级顺序将图分块加载到LLC中，并利用每一轮统计得到的任务与数据分块的关联信息，触发当前当前分块的关联任务批量执行，实现了对共享数据的高效访问；

其次，它针对计算共享难题，提出了一个基于核心子图的查询加速机制，它将传统的维护所有顶点索引值的“全局索引”瘦身为只维护高度顶点之间索引值的“核心子图索引”。两个高度顶点之间的最佳路径可能有很多跳，而核心子图相当于给所有互通的高度顶点增加了一条跳边，边的长度就是两点之间的最佳距离。这样，当查询到一个高度顶点时，程序就可以像访问邻居节点一下，访问所有其它的高度顶点，从而实现重叠路径的计算共享。瘦身后的核心子图索引开销远小于全局索引，从而可以选择更多的高度顶点加入到核心子图中，增大了高频共享路径的覆盖范围，提高了计算共享的性能。

最后，GraphCPP还通过预测不同查询任务的遍历路径，优先调度高度重叠的查询任务批量执行，进一步提高了并发查询的性能。

In response to the aforementioned challenges, we have designed GraphCPP, a data-driven system tailored for concurrent point-to-point queries on dynamic graphs.

To address the issue of data sharing among concurrent tasks, we propose a data-driven caching execution mechanism that transitions from the conventional "task→data" scheduling approach to a "data→task" strategy. This shift enables concurrent access to graph structure data across multiple tasks.Under this execution paradigm, GraphCPP initially determines the order of data scheduling, segmenting graph structure data into fine-grained blocks at the LLC level. Subsequently, it associates each query task with the relevant graph block based on the block where the task's active vertex set resides. Given that the active vertices change in each round, the number of associated tasks for shared blocks is dynamically updated, with blocks having more associated tasks receiving higher scheduling priority. To implement the "data→task" scheduling approach, GraphCPP leverages an associated task triggering mechanism. This mechanism prioritizes the loading of graph blocks into the LLC and uses task-data association information obtained in each round to trigger batch task execution associated with the current block, resulting in enhanced and efficient access to shared data.

In response to the challenge of computational sharing, GraphCPP introduces a query acceleration mechanism based on core subgraphs. This mechanism streamlines the traditional "global index," which maintains index values for all vertices, into a "core subgraph index" that exclusively manages index values between high-degree vertices. The core subgraph effectively creates direct edges between interconnected high-degree vertices, representing the best index between them. When querying a high-degree vertex, the program can access all other high-degree vertices, akin to visiting neighboring nodes. This approach enables computational sharing across overlapping paths. The streamlined core subgraph index incurs significantly lower overhead than the global index, allowing for the inclusion of more high-degree vertices in the core subgraph. This expansion broadens the coverage of frequently shared paths, ultimately enhancing computational sharing performance.

Furthermore, by predicting the traversal paths of different query tasks, we prioritize batch task execution for tasks with substantial overlap, further optimizing the performance of concurrent queries.

本文主要做出了如下贡献：

1. 分析了现有点对点查询系统处理并发点对点查询任务时冗余数据访问带来的性能瓶颈，并提出利用并发查询任务之间的数据访问相似性优化并发任务吞吐量。
2. 开发了GraphCPP，一个动态图上数据驱动的并发处理点对点查询系统，实现了并发任务之间的数据共享和计算共享，并提出了一个相似任务批量执行策略。
3. 我们将GraphCPP与当前最先进的点对点查询系统XXXXXX进行对比，结果表明XXXXXXXXX

This paper makes the following contributions:

1. Analyzed the performance bottleneck caused by redundant data access in existing point-to-point query systems when handling concurrent point-to-point query tasks. Proposed leveraging data access similarity among concurrent query tasks to optimize concurrent task throughput.

2. Developed GraphCPP, a data-driven concurrent point-to-point query system on dynamic graphs, achieving data and computational sharing among concurrent tasks. Additionally, introduced a strategy for batch execution of similar tasks.

3. We compared GraphCPP with the state-of-the-art point-to-point query system XXXXXX. The results demonstrate XXXXXXXXXX.

背景和动机

所需图像（还没画，占位）

1，统计各个场景的实际并发数，证明并发查询的需求。也可以用数据的形式展现，不需要图像）

1. 统计不同系统并行查询执行时间（保证总任务数为1000，单次并发数目越多，整体的计算时间变化），说明并行执行效率很差。
2. 统计重叠数据访问占总数据的比例（重叠数据应该是任务之间访问数据的交集，并发数目越多，重叠数据的比例应该越大），证明“数据冗余访问”。
3. 统计并行调度缓存错失率（调整并发任务数量，并发数越大，换错措施率越高），说明并行调度的方案低效的原因。
4. 统计计算高度顶点占总遍历数据的比例（高度顶点占顶点数的比例很低，但是占访问路径的比例很高）。
5. 统计高度顶点占冗余数据的比例（调整高度顶点的度数阈值，阈值越低，占比应该越高），证明不同任务会重复计算高度顶点之间的距离。
6. 统计固定数目的全局顶点的索引覆盖率和开销。

现有的解决方案聚焦于加速单次查询的速度，如PnP使用基于下界的剪枝方法来减少查询过程中的冗余访问；Tripoline通过维护中心顶点到其它顶点的日常索引，实现无需先验知识的快速查询；SGraph利用三角不等式原理，提出了基于“上界+下界”的剪枝方法，进一步减少点对点查询过程中的冗余访问；然而如图1（统计各个场景的实际并发数，证明并发查询的需求。也可以用数据的形式展现，不需要图像）

）所示，我们的统计表明，图上的并发点对点查询正在成为原来越迫切的需求，它们更重视并发查询任务的吞吐量，对于单次查询的速度则比较宽容。如图2（统计不同系统并行查询执行时间，说明并行执行效率很差。）所示，我们证明现有系统在处理大规模并发查询时吞吐量很差。这种坏结果出现的原因是并发任务之间存在对图结构数据大量的冗余访问。为了定性地分析上述问题，我们在XXXXX（机器配置），选取了XXXXX（现有最佳方案），在XXXXX（图数据集上），进行并行点对点查询的性能评测。

本章分为三个部分，我们首先介绍了并发点对点查询中的一些概念；其次分析了当前点对点查询方案处理并发任务时的性能瓶颈；最后展示了我们根据观察分析获得的启发。

BACKGROUND AND MOTIVATION

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

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

Preliminaries

定义一：图：我们使用G=(V,E)来表示有向图，其中V是顶点的集合，E是由V中顶点组成的有向边的集合（无向图中的边可以被拆分为两个不同方向上的有向边）。我们使用|V|，|E|分别表示顶点的数目以及边的数目。

定义二：图分区：我们使用Pi=(VPi,EPi)来表示有向图的第i个图分区，使用VPi表示图分区中顶点的集合，EPi是由VPi中顶点组成的有向边的集合。对于分布式系统，不同机器上的图分区Pi各不相同，我们采用边切分的方式划分图，同一个顶点可能出现在不同计算节点上，但是只有一个主顶点，其它的都是镜像顶点。

定义三：点对点查询：我们使用qi=(si，di)表示任务i对应的查询。其中si和di分别表示查询qi对应的源顶点和目的顶点。查询qi得到的结果值为Rsd，对于不同的算法，它有着不同含义，例如对于最佳路径查询Rib表示si和di之间的最佳路径。我们使用Q={q1,q2,…q|Q|}表示并发的点对点查询集合，其中|Q|表示查询的总个数。

定义四：索引：索引记录了某个顶点到其它顶点的距离，它通过频繁访问的路径的与计算实现了计算共享。全局索引：选取图中度数最高的k个顶点作为索引顶点hi（i∈[1,k]，k值由用户指定，一般设为16），di,j（Vj∈V）表示从索引顶点hi出发到达图中任意顶点Vj的距离，当两点之间不存在可达路径，该值设为极大值。同理dj,i（Vj∈V）表示从图中任意顶点Vj出发到达索引顶点hi的距离。无向图的di,j和dj,i是相等的。核心子图索引：选取度数排名在（k,m）区间的高度顶点hj建立顶点（j∈(k, m]，m值由用户指定，一般比k大一个数量级以上）。

定义五：上界和下界：在点对点查询中，上界UB表示当前已知的从源点到目的顶点的最佳路径的索引值，下界则LB表示从当前顶点v到目的顶点保守的最佳路径索引值，预测的LB小于或等于顶点v到目的顶点实际的收敛值。根据图上的三角不等式，如果一条路径的距离大于UB，或者加上LB的值后比UB大，则这条路径一定比已有的路径差，需要被剪枝。上下界的值需要借助索引来推导出，它们本质上是一种计算共享。

**Preliminaries**

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

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

Definition 3: Point-to-Point Query. We use q\_i=(s\_i,d\_i) to represent the query corresponding to task i. Here, s\_i and d\_i respectively denote the source and destination vertices of query q\_i. The result value obtained by query q\_i is represented as R\_{sd}. For different algorithms, it holds different meanings. For example, for best path queries, R\_{ib} represents the best path between s\_i and d\_i. We use Q={q\_1,q\_2,\ldots,q\_{|Q|}} to represent the set of concurrent point-to-point queries, where |Q| denotes the total number of queries.

Definition 4: Index: An index records the index from one vertex to other vertices and achieves computational sharing by calculating frequently accessed paths. Global Index: We select the top k vertices with the highest degrees in the graph as index vertices hi (where i∈[1, k], and users can specify k according to their needs, typically set to 16). In this context, di,j (where Vj∈V) represents the index from index vertex hi to any vertex Vj in the graph. If there is no reachable path between two vertices, the value is set to an extremely high value. Similarly, dj,i (where Vj∈V) represents the index from any vertex Vj in the graph to the index vertex hi. In undirected graphs, di,j and dj,i are equal. The creation of this global index is designed to meet specific requirements. Core Subgraph Index: We choose highly connected vertices hj in the degree range (k, m) to establish a core subgraph index (where j∈(k, m], and users can specify m based on their requirements, typically one order of magnitude larger than k).

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

定义六：核心子图：和索引类似，核心子图也会筛选出图上的高度顶点，但是它筛选的阈值更低，意味着有更多顶点可以被选中。这些高度顶点彼此相连组成核心子图，图上两个高度顶点之间边的权重代表两个点之间的索引值，倘若两个顶点最终不可达，则边的权重为极大值。核心子图和全局索引的重要区别是核心子图只记录高度顶点之间的索引值，并不记录达到非高度顶点的索引值。

并发点对点查询任务的性能瓶颈

在本节，我们改编出了当前最先进的点对点查询系统SGraph的并发版本SGraph-C，并运行社交网络图数据集-twitter，以探究现有系统处理并发任务的性能瓶颈及其原因。我们发现现有解决方案并发执行时的两大瓶颈：冗余数据访问开销、冗余的计算开销。

**并发任务的****冗余数据访问开销**

并发点对点查询任务在同一个底层图上执行图遍历，它们的遍历路径有很大一部分重叠。如图3，我们的数据表明……。然而在传统的“任务->数据”调度模式下，不同任务会在不同时刻独立将自己所需要的图数据分块加载到缓存处理，导致在缓存中保存相同图分块的多个副本。由于缓存的空间限制，随着并发数目的增大，任务之间的资源竞争导致严重的缓存未命中损失。如图4……。

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

**Performance Bottlenecks in Concurrent Point-to-Point Query Tasks**

In this section, we present our adapted concurrent version of the state-of-the-art point-to-point query system, SGraph-C, and run it on the Twitter social network graph dataset to investigate the performance bottlenecks and their underlying reasons in existing systems when dealing with concurrent tasks. We identify two significant bottlenecks in concurrent execution: redundant data access overhead and redundant computational costs.

Redundant Data Access Overhead in Concurrent Tasks

Concurrent point-to-point query tasks perform graph traversal on the same underlying graph, with a substantial overlap in their traversal paths. As depicted in Figure 3, our data indicates... However, in the traditional "task->data" scheduling model, different tasks independently load the graph data blocks they need into the cache for processing at different times, resulting in multiple copies of the same graph blocks being stored in the cache. Due to cache space limitations, as the number of concurrent tasks increases, resource contention between tasks leads to severe cache misses. As shown in Figure 4...

**并发任务的冗余计算开销**

由于图数据的幂律分布特性，少数高度顶点连接了大部分边。因此，如图5所示，尽管高度顶点仅占总顶点数的一小部分（XX%），它们却出现在许多路径中（XX%）。进一步的分析，如图6所示，揭示出不同任务访问的冗余路径数据中有相当比例的高度顶点。这意味着不同的查询任务会重复遍历高度顶点之间的最佳路径。在一个图快照周期内，高度顶点之间的查询路径是恒定的，因此对其进行重复计算是冗余的。此外，由于高度顶点拥有大量出边和入边，计算它们之间的最佳路径会导致巨大的计算负担。

一些现有的解决方案尝试建立全局索引以减少不同任务的冗余计算。然而，如图7所示，全局索引面临着图中顶点覆盖率与索引固有开销之间的权衡问题。具体来说，当索引数量较少时，索引覆盖的路径也较少，无法实现较高的共享率。当索引数量增多时，与之相关的计算、存储和维护开销会成比例增加，降低了索引的好处。考虑到不同图数据集的属性差异以及并发查询场景的演化，确定合适的索引数量变得复杂。因此，全局索引无法有效解决计算冗余问题。

Redundant Computational Costs in Concurrent Tasks

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

我们的启发

通过上述的观察，我们得到到了以下启发：

**观察**1：不同任务之间存在数据访问相似性，它们的遍历路径有很大部分是重叠的。但是由于不同任务访问重叠数据的时间不同，且现有的点对点查询系统并不支持任务之间的数据共享，对重叠数据的访问成了冗余开销。这启发我们开发高效地细粒度数据共享机制，通过支持不同任务在不同时间对相同数据进行访问共享，来减少数据访问开销，提高并发查询的吞吐量。

**观察2**：高度顶点组成的路径段更可能被不同的任务重复遍历。不同的查询路径可以看做一条条线，高度顶点就是这些线段的交点，会频繁出现在不同的任务中。现有的全局索引方式开销巨大，往往对索引顶点数设限，导致可共享的路径占比很低。这启发我们通过轻量级的索引实现更好的计算共享。

**Our Motivation**

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

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

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

系统概述

系统架构

为了提高并发点对点查询的执行效率，在对并发点对点查询的计算细节进行仔细研究后，我们提出了一个数据驱动的高效并发点对点查询系统-GraphCPP。我们系统的关键思想是实现并发点对点查询任务之间的数据共享和计算共享。如下图所示，为了实现这个目标，GraphCPP提出了一个高效地数据驱动的缓存执行机制，利用并发任务之间的数据相似性，实现了重叠数据的共享访问。以及一个基于核心子图的查询加速机制，共享不同查询任务对相同热路径的计算。此外，它还通过预测不同查询的遍历路径，驱动路径重叠的相似查询批量执行，进一步利用了数据相似性。

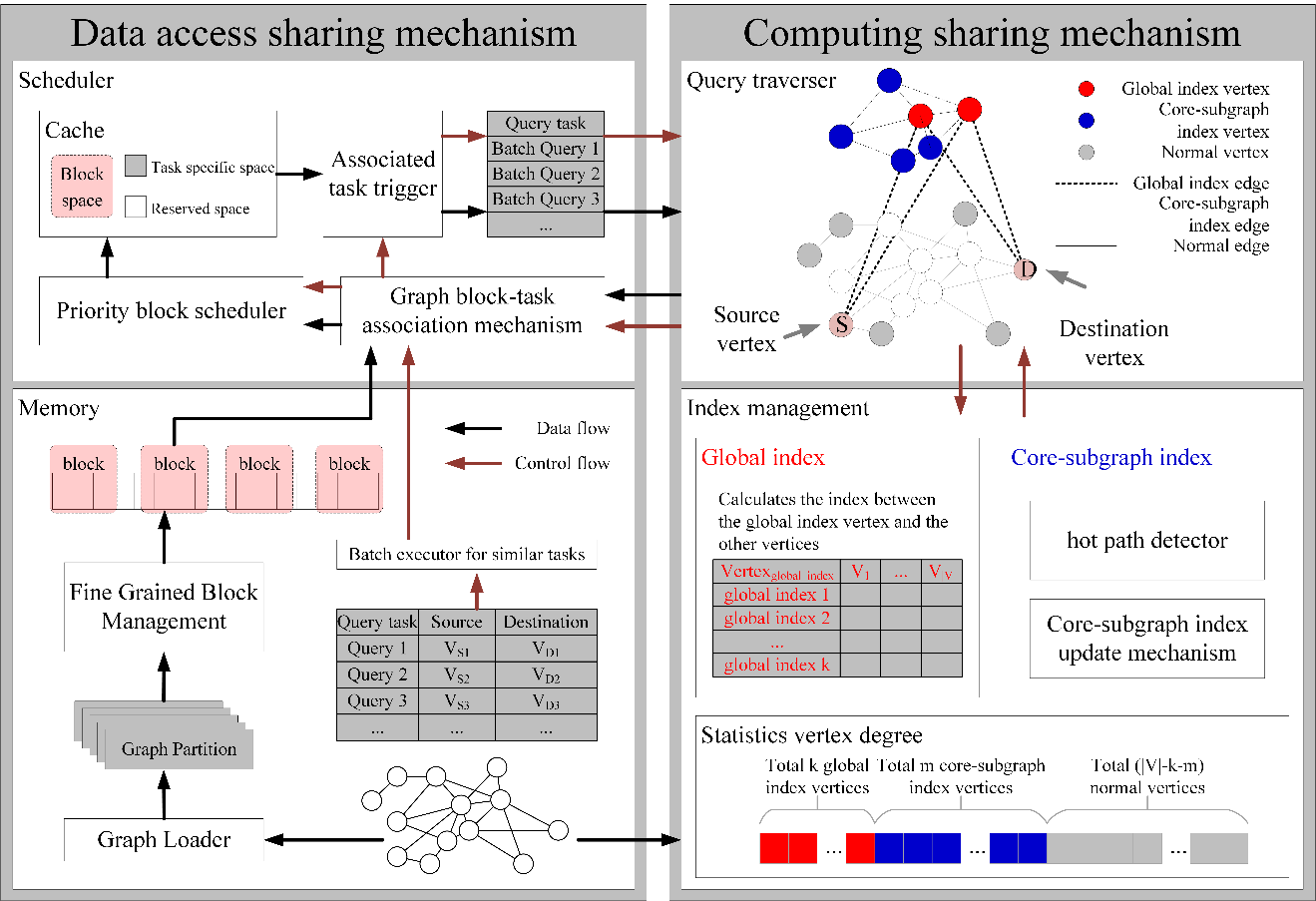
数据访问共享机制。它负责利用并发任务的数据访问相似性实现图结构数据的细粒度共享。首先，它像其它分布式图计算系统[xx]一样将原始图数据划分为粗粒度的图分区，交由不同的机器并行处理。然后使用一个细粒度的分块管理器将粗粒度的图分区划分为细粒度的图分块。接着执行图分块-任务之间的关联机制：假如一个查询任务qi在某个图分块bi存在活跃顶点，那么就认为qi和bi之间存在关联关系。分块优先级调度机制会根据不同图分块包含关联任务的数量，优先调度有更多关联任务的图分块到LLC。关联任务触发器会根据LLC中活跃图分块信息，以及图分块-任务的关联信息，筛选出所有有关联关系的任务在共享图分块上批量执行。

计算共享机制。它负责借助索引信息（包含全局索引和核心子图索引）利用并发查询任务之间的计算相似性实现路径计算的共享。1）**预处理阶段**：它首先统计原始图数据中所有顶点的度数信息， 将原始图的顶点按照度数由大到小排序，选择度数排名位于[1,k]的顶点作为全局索引顶点，选择度数排名位于[k+1,k+m]的顶点作为核心子图索引顶点；2）**索引建立阶段**：一方面，在查询开启前执行点对多点查询算法计算全局索引顶点与图中所有顶点的最佳路径值（例如对于PPSP任务，需要执行SSSP算法），对于有向图，需要分别计算全局索引的出边路径值和入边路径值，对于无向图则只需要计算一次；另一方面：GraphCPP通过一个运行时方法来动态维护核心子图。具体地，一个顶点对的最佳路径的任意一段子路径，都是对应顶点之间的最佳路径。因此核心子图无需对热路径进行预计算，而是在每次查询结束后，执行热路径探测机制发掘一条最佳路径中的热路径，并将其加入到核心子图结构中。3）**计算共享阶段**：全局索引顶点一般充当大量路径的中介枢纽顶点，因此在查询开始时，使用全局索引顶点可以计算出一条可达路径的路径值。这条路径不一定是查询顶点对之间的最佳路径，但是可以对剪枝查询提供参考。此外在基于上界+下界的剪枝遍历中，还可以借助全局索引预估查询路径的路径值，更早对不合适路径进行剪枝。使用全局索引是第一层次的计算共享；核心子图维护了热顶点之间的最佳路径值，它充当了查询任务之间的高速公路。当查询任务遍历到核心子图中的热顶点，相当于接入了高速公路，它可以借助核心子图从入口顶点快速到达核心子图的出口顶点，而无需重复计算这段热路径的路径值。使用核心子图是第二层次的计算共享。

GraphCPP Overview

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

The data access sharing mechanism is responsible for partitioning the graph structure data into fine-grained blocks, selecting shared graph blocks, and triggering the execution of associated tasks in batches. First, like other distributed graph computing systems, it partitions the original graph data into coarse-grained graph partitions for parallel processing on different machines. Then, a fine-grained block manager is used to further divide the coarse-grained graph partitions into fine-grained graph blocks. Next, based on the partition where the active vertices of the query task are located, an association mechanism between graph blocks and tasks is established. The graph block priority scheduling mechanism will prioritize scheduling graph blocks with more associated tasks to the Last Level Cache (LLC) based on the number of associated tasks in different graph blocks. The associated task trigger, based on the active graph block information in the LLC and the association information between graph blocks and tasks, selects all tasks with association relationships to be executed in batches on the shared graph blocks.

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

整体执行流程

我们将以伪代码形式展示GraphCPP的整体执行流程。该算法接收两个输入参数：当前计算节点所包含的所有图分块的集合B以及当前计算节点所包含的所有查询任务的集合Q。首先，我们分配一个动态大小的连续内存空间，用于存储所有的查询任务（第一行）。然后，我们进入一个循环处理过程，只要仍有未结束的查询任务（第二行），GraphCPP将调用ChoseNextSharingBlock来更新查询任务与图分块之间的关联关系，并选择当前优先级最高(有着最多关联任务)的图分块bi。通过统计每个任务的关联分块（即任务在当前分块存在活跃顶点），我们可以确定与当前图分块bi相关联的所有查询任务（第四行）。接下来，我们将bi加载到缓存，并并行处理所有相关联的查询操作qi（第五行）。我们调用GraphCPPCompute在

Overall Execution Workflow

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

当前分块上执行点对点查询操作qi。如果查询尚未结束，我们会更新查询qi的状态，生成新的查询任务（第六行）。如果新生成的查询与当前的图分块bi存在关联，将qi添加到Qbi，然后返回第五行以继续查询。否则，将新生成的查询信息保存到查询任务集合中，任务被挂起。

|  |
| --- |
| Algorithm 1: Concurrent Point-to-Point Queries on a Set of Graph Blocks Owned by a Graph Partition. |
| 1: MallocBuffers( *B*, *Q* ) //*B* is the set of graph blocks, and *Q* is the set of query tasks  2: While has\_active( *B* ) do：  3: *bi* ← ChoseNextSharingBlock( )  4: *Qbi* ← ChoseAssociatedQueries( *bi* )  5: Parallel\_for\_each *qi* ∈ *Qbi* do: // Execute queries in *Q* in parallel, which is associated with block *bi*  6: new\_query =GraphCPPCompute( *qi*, *bi* ) // The implementation function for point-to-point queries returns the active vertex set after one round of task iteration.  7: if(has Associated( ( *bi* , new\_query ) ):  8: *Qbi*.Push( new\_query )  9: else:  10: *Q*.Push( new\_query ) |

上述算法展示了GraphCPP中的数据共享机制，其中的GraphCPPCompute函数则使用了计算共享机制。下面的章节将详细介绍两个优化机制。

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

数据访问共享机制

在2.2节中我们观察到并发任务之间的图结构数据访问存在很大一部分重叠，在现有处理机制下，这部分重叠数据并不能被共享利用。而对于图上的点对点查询任务来说，数据的访问顺序并不会影响结果的正确性。我们的数据共享机制本质上是将原本的“任务→数据”线性任务调度顺序，改为“数据→任务”细粒度并发任务调度顺序，从而提高缓存利用效率，提高系统吞吐量。接下来我们从两个问题出发描述如何实现数据访问共享，并在最后描述了一个进一步利用数据访问相似性的措施。

一、如何确定共享的数据部分？

1，确定共享图分块粒度。分布式内存图计算系统需要将数据载入缓存以提升数据访问效率，所以理想情况下共享图分块的数据要能完整地载入LLC，从而避免访问分块不同部分带来的频繁换入换出。但是图分块的粒度也不能过于小，否则会增加任务处理的同步开销。我们使用公式x来确定合适的共享图分块大小。其中BS表示待确定的共享图分块的图结构数据的大小，GS表示分块所属的图分区的图结构数据的大小，|V|表示分区上图的顶点总数，VS表示存储一个顶点的状态信息平均所需的空间大小，N表示并发查询的任务数，LLCS是LLC缓存空间的大小，RS是预留的冗余空间的大小。公式x右侧的两项分别表示图结构数据和任务特定数据（其大小与图分块的规模和并发查询任务数成正比）。公式右侧表示减去缓存预留空间后，每个任务剩下的可使用空间的大小。通过这个公式，我们求得了在适应LLC容量前提下，每个共享图分块的最大粒度。

**Data Access Sharing Mechanism**

In Section 2.2, we observed a significant overlap in the graph structure data access among concurrent tasks. Under the existing processing mechanism, this overlapping data cannot be shared and utilized. However, for point-to-point query tasks on the graph, the order of data access does not affect the correctness of the results. Our data sharing mechanism essentially transforms the original "task → data" linear task scheduling order into a "data → task" fine-grained concurrent task scheduling order, thereby improving cache utilization efficiency and system throughput. Next, we will describe how to achieve data access sharing starting from two aspects, and conclude with an additional measure that further leverages data access similarity.

A. How to Determine Shared Data Segments?

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

2，逻辑划分。确定好共享图分块的粒度后，GraphCPP就可以在图预处理的过程中，采用逻辑划分的方式，将分布式系统上粗粒度的图分区划分为细粒度的共享图分块。清单x展示了GraphCPP划分图分块的伪代码：

|  |
| --- |
| Algorithm: Logical Partition Algorithm. |
| 1: function Partition(Pi , B) // *B* is the set of graph blocks owned by graph partition Pi.  3: block\_table = null  4: for each e ∈ Pi do: //e is an edge in partition Pi  5: if e.src in block\_table:  6: block\_table[e.src]++  7: else:  8: block\_table[e.src]=1  9: end if  11: if block\_ table.size（）≥ SC:  12: B.push(block\_table)  13: block\_table.cear( )  14: end if  15: end for |

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

逻辑分区函数接收两个参数，一个是以边表形式记录的图分区结构数据Pi ，一个是该分区所拥有的图分块集合B。我们通过一个字典结构block\_table来统计图分块信息，它的key记录边的源顶点ID，value记录该顶点对应的出边的数目。循环遍历分区中的每一条边。如果该边已经被加载到当前的分区，将分区对应的出边数量加一。如果该顶点是第一次加入到字典中，将分区的出边数置为1。每次遍历完一条边都会判断当前分块是否已满，若分块已满，将当前分块加入block\_set。这样当分区中的所有数据遍历完一遍，分区的每一条边都被划归到某一个图分块，我们就得到了从逻辑上划分的图分块的集合。

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

二、如何实现多任务间的数据共享

1，建立共享分块-查询任务的关联。通过之前的步骤，我们用逻辑划分的方式，实现了细粒度的图分块。由于只是逻辑上的分块，数据在物理存储介质上依然是连续的，所以可以通过顶点的ID轻松判断出顶点所在的分区。执行查询时，每个任务*qi*在迭代计算过程中会维护一个活跃顶点集Setact,i，它遵循以下更新策略：1，初始时Setact,i仅包含查询源顶点Si。2，按照点对点查询算法的流程处理Setact,i中的活跃顶点，处理后的顶点会被从活跃顶点集中移除。 3，如果一个顶点的状态在本轮中被改变，且它没有被剪枝，则该顶点被加入到Setact,i等待下一轮处理。我们首先通过顶点的ID号反推出其所在的图分块，然后利用专门设计的数组存放每个任务所遍历的分区。由于点对点查询采用基于剪枝的遍历策略，每一轮执行中活跃顶点的数量并不多，所以可以以较低的开销建立查询任务与所属分块的关联。

2，确定分区调度的优先级。建立好查询任务与所属分块的关联后，我们可以统计到每个分块关联的任务数量。任务数量越多，代表共享该分块的任务越多，此时该任务带来的收益越大，优先将该分块调入LLC中。

3，触发关联任务并发执行。我们已经获得了共享图数据分块，根据共享分块-查询任务的关联关系可以推导出活跃的查询任务，它们共享LLC中的图结构数据，我们采用批量计算的方式执行这些查询任务。如算法X所示，活跃任务执行一轮后会产生新的活跃顶点，倘若新的活跃顶点仍然与当前的共享分块相关联，查询任务会继续执行。共享分块会始终停留在LLC，直到与该分块关联的所有查询任务都被处理完毕，才会换出。

B. Achieving Data Sharing Among Multiple Tasks

1. Establishing the Association between Shared Blocks and Query Tasks. Through the previous steps, we have achieved fine-grained graph partitioning using a logical approach. Since this is only a logical partitioning, the data remains contiguous on the physical storage medium. Hence, it is easy to determine the partition a vertex belongs to based on its ID. During the execution of a query, each task qi maintains a set of active vertices, Setact,i, following the update policy outlined below:

a. Initially, Set\_(act,i) contains only the source vertex Si of the query.

b. The active vertices in Set\_(act,i) are processed according to the point-to-point query algorithm, and the processed vertices are removed from the set of active vertices.

c. If a vertex's state is changed in this round and it is not pruned, the vertex is added to Set\_(act,i) and awaits processing in the next round.

We first deduce the graph block in which the vertex is located based on its ID, and then use a specially designed array to store the partitions traversed by each task. Since point-to-point queries employ a pruning-based traversal strategy, the number of active vertices in each round is not large. Therefore, it is possible to establish the association between query tasks and their corresponding blocks with relatively low overhead.

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

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

相似任务批量执行

不同查询任务随机到来，它们的遍历路径也有很大的不同。我们发现当两个任务的相似程度过低，它们之间的重叠路径比例也会降低，甚至可能没有重叠部分。而如果两个查询的起始顶点和目的顶点都处于临近的图数据分块，它们在查询过程中的遍历路径也大概率是临近的。对此我们提出了一个相似任务感知的批量执行策略，每次从任务池中筛选相似任务批量执行，以进一步地利用数据相似性。具体地，GraphCPP首先从任务池中随机选择一个查询任务，获取任务的起始顶点和目标顶点。然后执行k跳SSSP获取起始顶点的邻居顶点集SetS，以及目标顶点的邻居顶点集SetD（k的大小由用户确定，默认设为3）。随后遍历任务池，筛选出所有起始点位于SetS，目的点位于SetD的查询任务，它们被作为相似任务并发处理。需要注意的是，如果某个查询的起始顶点或目的顶点属于高度顶点，可以直接使用索引来加速查询过程，无需使用常规的查询步骤。排除掉高度顶点后K跳SSSP本身的开销很小，且执行过程可以和正常查询并发执行，执行开销可以忽略不计。**Batch Execution of Similar Tasks**

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

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

计算共享机制

GraphCPP通过全局索引机制和核心子图索引机制实现了两个层次的计算共享。全局索引的固有开销很大，因此实践中往往选择很少量的热顶点建立全局索引。由于幂律分布的特点，这些热路径充当了不同查询的中介枢纽节点，虽然很难确保途径热顶点的路径一定是查询的最佳路径，但是它们为剪枝查询提供了参考值，实现了第一层次的计算共享；进一步的，核心子图机制在无需预处理的前提下，挖掘已有查询结果的最佳路径，实现对热路径的计算共享。和全局索引相比，核心子图更加的轻量级，因此可以通过增加热顶点数目做到更高的覆盖范围，提供更精确的上界值。算法XXXX展示了核心子图查询的伪代码。

**Computation Sharing Mechanism**

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

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

|  |
| --- |
| Algorithm: Shared Computation |
| 1: function IndexPreprocess(V, k, m):  2: globalVertices, coreSubgraphVertices = SortVerticesByDegree(V, k, m)  3: globalIndex = BuildGlobalIndex(k)  4: coreSubgraphIndex = InitializeCoreSubgraphIndex(m, globalIndex)  5: function SharedComputation(globalIndex, coreSubgraphIndex, query):  6: bound = FirstLevelSharedComputation(globalIndex, query)  7: while (activeVerticesCount != 0):  8: activeVertex = GetNextActiveVertex()  9: if activeVertex in coreSubgraph:  10: UpdateBounds(bound, SecondLevelSharedComputation(coreSubgraphIndex, query))  11: for neighbor of activeVertex:  12: UpdateBoundsByNeighbors(neighbor)  13: activeVerticesCount = UpdateActiveVertices()  14: function UpdateCoreSubgraph(bestPath):  15: hotPath = ExtractHotPath(bestPath)  16: hotPathValue = CalculateHotPathValue(hotPath)  17: AddToCoreSubgraph(hotPath, hotPathValue) |
|  |

实现计算共享的执行步骤如下：1，索引预处理（第1-4行），系统在对顶点的度数进行排序之后，选择度数最高的k+m个热顶点，将前k个顶点作为全局索引顶点（k值由用户确定），剩下的顶点作为核心子图顶点。全局索引的计算在预处理阶段完成，GraphCPP会执行SSSP算法计算k个高度顶点与图上的所有顶点的最佳路径（包含索引值和路径父节点），将结果存入以高度顶点id为索引的数组保存。核心子图省去了预计算过程，直接复用每次查询的计算结果，因此在预处理阶段只需要进行初始化。2，计算共享（第5-13行）：全局索引顶点充当了查询路径的枢纽节点，大部分查询都存在至少一条经过全局索引顶点的路径，虽然这条路径不一定是最佳路径，但是却为查询剪枝提供了一个可靠参考。因此每次执行点对点查询前，首先借助全局索引确定粗略的界限值，这是第一层次的计算共享。随后执行迭代查询算法，不断处理新的活跃顶点，直至所有顶点都收敛。对于每一个活跃顶点，我们判断它是否属于核心子图。初始时核心子图为空，所以不会参与共享，随着查询任务的执行，核心子图中逐渐新增了更多热路径。当活跃顶点属于核心子图成员时，就可以借助核心子图直接获取到对应起始顶点的热路径值，从而避免了重复计算。3，更新核心子图（第14-17行）：为了保证核心子图的轻量级，我们不对热路径进行预计算，而是通过从已有的最佳路径中发掘热路径的子集，来复用之前的计算结果。显然一条最佳路径上任意两个顶点之间的路径都是最佳路径，因此我们只需要以很小的开销，识别已有结果中的热顶点，并通过前缀和方法求得热顶点之间的结果即可。为了实现这一目的，我们需要在查询过程中保留遍历路径以及从源顶点出发到达每一个中间点的路径值，由于点对点查询本来需要计算这些信息，因此并不会造成很大的开销。通过上述步骤，我们用轻量级的核心子图索引，实现了高效地数据共享。

The execution steps for achieving computation sharing are as follows: 1. Establish a Global Index (Lines 1): We employ a strategy similar to SGraph for computing the global index. After sorting the degrees of vertices, the system selects the top k vertices with the highest degrees (where the value of k is user-determined). Subsequently, an SSSP algorithm is executed to compute the best paths (including index values and path parent nodes) between these k high-degree vertices and all vertices in the graph. The results are stored in an array indexed by the IDs of the high-degree vertices. 2. Establish a Core Subgraph Index (Lines 2): Allowing more high-degree vertices, typically one order of magnitude larger than k, which are ranked in the top m in terms of degree, to be included in the core subgraph. As the global index vertices have already recorded the indices to reach the global index, these vertices are excluded. Additionally, once the global index is established, point-to-point queries for the best paths between points on the core subgraph can be directly computed using upper and lower bound pruning. 3. Query Acceleration (Lines 3): Perform point-to-point queries, starting by utilizing the global index to determine approximate upper and lower bounds. Subsequently, pruning queries begin. Under normal circumstances, the system traverses each outgoing edge vertex of the current vertex, sequentially performing pruning checks on the index values of each vertex to determine the next round of active vertices. If the current query vertex belongs to the core subgraph, in addition to visiting neighboring out-edge vertices, all other high-degree vertices connected to this vertex must also be accessed. Under normal circumstances, the state propagation between these high-degree vertices may require multiple hops. With the core subgraph, the propagation between these points can be accomplished in a single step. In addition to expediting state propagation, a hidden factor is that the core subgraph is populated with high-degree vertices, making them more likely to appear on the best path between two points, thereby expediting the path discovery process. 4. Query Termination (Lines 6): Apply upper and lower bound query techniques for pruning. For unidirectional queries, starting from the source vertex, reaching the destination vertex indicates the discovery of a path. For bidirectional queries, the convergence of queries from both directions indicates the discovery of a path. If a new path value is smaller than the current upper bound, it is updated as the new upper bound. If the path value is greater than the current upper bound, it is pruned. The discovery of a path does not imply the end of iteration; it is necessary to assess the active vertices in the graph. Only when all possible paths have been attempted, the upper bound is updated to the best path value, and all vertex edge path values are greater than the current upper bound, and the number of active vertices decreases to zero, does the iteration conclude. Through the aforementioned steps, we achieve efficient data sharing using the lightweight core subgraph index.

更新机制

实际应用中，供查询任务遍历的底层图常常是动态变化（发生边增加eadd和边删除edelete）的，而变化的图结构数据会导致索引值产生误差。因此，当动态图发生图更新时，我们除了需要更新图结构信息，还需要动态更新索引。图结构信息更新：GraphCPP通过邻接列表来存储每个顶点的出边邻居，因此我们只需要根据边增加（边删除）时的源顶点信息，修改对应的出边邻居的邻接表就可以了。索引更新：我们采用一种增量更新的方式，依次更新全局索引和核心子图索引，尽可能减少了索引更新的冗余计算开销。

全局索引顶点数目较少（有k个全局索引顶点），但是记录的索引值非常多（有k\*|V|个全局索引值），所以可以把索引信息存放在各个顶点上。每个顶点维护两个表格，table1记录了该顶点到k个全局顶点的最佳路径上的父节点，table2该顶点到k个全局顶点的最佳路径的索引值。我们根据边更新的类型，来对这两个表格进行增量更新。具体地，当发生边增加更新eadd时，我们首先获得更新边的源顶点src，目的顶点dst，以及两点之间的权重。接着我们依次检查每个全局索引顶点，如果Indexsrc+weight>indexdst，则更新table1中的parentdst为src，table2中的indexdst为Indexsrc+weight,否则无需更新该全局顶点索引；当发生边删除更新时，我们依次检查每一个全局索引顶点，判断parentdst是否等于src。如果是，表示我们删除了原本到达dst的最佳路径，此时需要重新计算indexdst。同时和其它的增量计算方法一样，对dst的更新会逐渐向外传递，所有经过dst的最佳路径的下游顶点都需要被更新。如果parentdst不等于src，则无需更新该全局索引顶点。

核心子图索引只记录少量高度顶点之间的索引，最多只需维护m\*m个索引值（m的数量级远小于图数据规模），所以我们采用独立的二维数组来存储核心子图。具体地，边增加更新在原有的图结构上做加法，因此可能产生新的捷径，导致原有的最佳路径退化为非最佳路径。对于我们的剪枝查询来说，非最佳路径的索引会导致早期的界限值偏大，但是随着迭代的进行，点对点查询会依然会遍历到更优路径，最终会收敛到正确的最佳路径。此时从收敛路径中提取最新的热路径值，即可完成对热路径的更新；对于边删除更新，GraphCPP会判断删除边的两个顶点是否同时出现在某个热路径上。如果是，原有的热路径被中断，所有受影响的热路径都失效。如果只有一个顶点或者没有顶点出现在某个热路径上，则删除的边不会影响对热路径产生影响，此时无需更新。由核心子图索引复用了每次查询的最佳路径结果，无需单独计算 ，因此整体开销较小。

上述机制实现了对图结构数据、全局索引、核心子图索引的增量式维护。考虑到细微的图更新不会对整体计算结果产生大的影响，我们暂存细微的图更新∆G直到它的大小超出预设的阈值或者达到一定的时间间隔，才会执行批量图更新操作，从而进一步降低了更新开销。

Update Mechanism

In practical applications, the underlying graph traversed by query tasks often undergoes dynamic changes, involving edge additions (e\_add) and edge deletions (e\_delete). The changing graph structure data can lead to errors in index values. Therefore, when dynamic updates occur in the dynamic graph, we not only need to update the graph structure information but also dynamically update the indices.

Graph structure information update: GraphCPP stores the outgoing neighbor vertices of each vertex using an adjacency list. Thus, we only need to modify the adjacency list of the corresponding outgoing neighbors based on edge additions (or deletions) and the source vertex information.

Index update: We adopt an incremental update approach, sequentially updating the global index and core subgraph index to minimize redundant computational overhead.

The number of global index vertices is relatively small (k global index vertices), but it records a significant number of index values (k\*|V| global index values). Therefore, we can store the index information on each vertex. Each vertex maintains two tables: table 1 records the parent nodes on the best paths to k global vertices, and table 2 records the index values of the best paths from this vertex to the k global vertices. Depending on the type of edge update, we perform incremental updates on these two tables. Specifically, when an edge addition update (e\_add) occurs, we first obtain the source vertex src, destination vertex dst, and the weight between the two points. Next, we sequentially check each global index vertex. If Indexsrc + weight > Indexdst, we update parentdst in table 1 to src and indexdst in table 2 to Index\_src + weight. Otherwise, there is no need to update the index of this global vertex. In the case of an edge deletion update, we sequentially check each global index vertex, determining whether parentdst equals src. If it does, it means we have removed the original best path to dst. In this case, we need to recalculate index\_dst. Similar to other incremental calculation methods, updates to dst will gradually propagate outward, requiring updates to all downstream vertices of the best paths passing through dst. If parentdst is not equal to src, there is no need to update this global index vertex.

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

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

实验评估

The experiments are conducted on a 8-node cluster. Each machine consists of 2 Intel Xeon E5-2680 v4 CPUs which has 14 physical cores，256 GB memory，and 35MB LLC. All nodes are interconnected through a Infiniband network(bandwidth of 300Gbps). All programs are compiled with gcc version 7.5.0, openMPI version 4.1.2 and openMP enable.

Table 2 shows the graph datasets used in our experiments and their properties. Twitter-2010 and Friendster are social network graphs. UK-2007-05 and Gsh-2015-host are large web-crawl graphs. We use five different algorithms as benchmarks: Shortest Path(PPsP), Breadth First Search(BFS), Connectivity, Widest Paht(PPWP), and Narrowest Paht(PPNP). These alogorithms are important pairwise graph queries because they 1) are widely used; 2) have different characteristics in the data access and 3) are usually used to form many important complex graph applications such as clustering, classification, and prediction.

To evaluate the performance, we sequentially or concurrently submit PPsP, BFS, PPWP, and PPNP in order until the specific number of jobs are generated. We set the parameters randomly for each job, even though these jobs may be the same graph algorithm. For example, the root vertices are randomly selected for the BFS jobs and the PPsP jobs. For concurrent submissions, the time intervals between consecutive submissions follow a poisson distribution with a default λ value = 16. All benchmarks are executed 10 times, and the experimental results are reported as average value.

We compared the query performance of GraphCPP with PnP, Tripoline, and SGraph. Since all of them are not open source, we re-implement their mechanism based on Gemini distributed graph processing framework. And bacause of none of these three systems directly support concurrent operations, we made some modifications to enable them to handle multiple jobs simultaneously. In the end, we compared the performance with two different modes of systems: SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C. Among them, the systems with the "-S" suffix sequentially handles the jobs, while the systems with the "-C" suffix concurrently handle the jobs. In the systems with the "-C" suffix, the concurrent jobs are managed by the operating system.

Preprocessing Cost

Table3 shows the preprocessing cost of GraphCPP, SGraph, and Tripoline. We can observe that GraphCPP takes little additional time than the other two system, so as to create the blocks by traversing the graphs once and generate core subgraph. The chunking procedure and core subgraph generation only resulted in an average increase of xx in preprocessing time. As shown in Table 4, the additional storage cost of GraphCPP is also small and accounting for only xx%-xx% of the original graph space overhead. In general, when the graph has larger maximum out-degree and lower average out-degree, the additional space overhead compared to the original graph's space overhead is higher. This is because that vertices with larger out-degrees store more copies in different blocks and the additional space overhead typically scales proportionally with the ratio of the number of vertices to the number of edges. Furthermore, the additional space cost will vary depending on the number of vertices chosen during the construction of the core subgraph. Table 4 shows the results obtained when setting the number of core-vertices to xx.

Overall Performance Comparison

Figure 9 shows the total execution time of xx concurrent jobs with different schemes. It can be observed that for all graphs, GraphCPP achieves shorter execution times (thus higher throughput) than the other schemes. Compared to SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C, GraphCPP achieves an average throughput improvement of approximately xx, xx, xx, xx, xx, and xx times, respectively. The improvement of throughput is achieved by reducing the data access cost 和核心子图带来的高效剪枝in GraphCPP.

为了评估数据访问成本，我们进一步地将总时间分成了图处理时间和数据访问时间。如图10所示，相比于其他系统，GraphCPP图数据访问占用的时间更少，并且随着图规模的增加该部分占用比例进一步减少。例如，对于Gsh-2015-host，GraphCPP的数据访问时间相比于其他系统分别减少了xx倍-xx倍。GraphCPP能达到如此效果的原因有两个：1）不同并发作业所需图数据的相同部分在内存中只需要加载和维护一个副本，这减少了内存占用；2）图数据块根据关联任务数量的多少划分优先级，并定期地加载到LLC中被作业重用。优先处理优先级更高（关联任务数更多）能有效降低LLC miss率，减少不必要的内存数据传输。

接下来，我们评估不同系统的LLC利用率，并将结果显示在图11中。从图中可以看出，GraphCPP的LLC miss率低于另外6个系统。在UK-2007-05中，GraphCPP的LLC miss率仅为xx，相比之下，SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C的LLC miss率分别为xx，xx，xx，xx，xx，and xx。这主要是因为在GraphCPP中，多个作业共享LLC中的一个图数据副本，能够更加充分地利用LLC，提高作业的数据局部性。

进一步地，我们还跟踪了这16个作业交换到LLC中的数据总量。一般来讲，并发执行模式（-C）会比顺序执行模式（-S）交换更多的数据到LLC中，这是因为并发作业之间没有数据共享，不同作业之间强烈的缓存干扰会导致图数据在LLC中频繁的换入换出，产生更多的冗余内存数据传输。如图12所示，GraphCPP的数据交换量比SGraph-S，PnP-S，和Tripoline-S少得多（在UK-2007-05中分别是三者的xx，xx，and xx）。这是因为GraphCPP充分利用了并发任务之间的数据访问相似性。

核心子图的效率

我们也评估了核心子图策略对GraphCPP性能的影响。在全局索引数设置为16的情况下，GraphCPP-128，GraphCPP-256，和GraphCPP-without分别是核心子图索引选择128，256个顶点和不使用核心子图索引的版本。在图13中，我们展示了这三个版本核心子图占用内存空间的大小。可以看到GraphCPP-128和Graph-CPP-256相比于GraphCPP-without只增加了极少的内存空间占用（三者都维护16个全局索引）。这是因为相对于全局索引保存到所有顶点的距离，核心子图顶点只保存与其他核心子图顶点的距离，而这只需很少的额外空间。

图14显示了GraphCPP-128，GraphCPP-256，和GraphCPP-without在16个作业上的总执行时间。我们可以观察到GraphCPP-256和GraphCPP-128总是优于GraphCPP-without，并且GraphCPP-256也比GraphCPP-128更快。在Friendster上，GraphCPP-256和GraphCPP-128的处理时间分别只有GraphCPP-without的xx和xx。这是因为增加的这些核心子图顶点能够有效地限制上界和下界，提高剪枝效果。

此外，我们还追踪了核心子图的维护时间。如图15所以，在图发生变化时，核心子图索引的维护开销只占全局索引开销的很少部分（GraphCPP-256和Graph-128分别只多xx和xx）。这是因为核心子图索引只需要更新到其他受图更新影响的核心子图顶点的距离。显然地，随着核心子图顶点数目的增多，其维护开销也会不断增加，但是对计算的好处不会按比例增加。根据我们的评估，将核心子图顶点数设置为图总顶点数的xx%（或者固定数目）能在效率和维护开销方面达到一个平衡。

扩展性

图16显示了不同PPsP作业并发数目下GraphCPP和另外6个系统的性能比较。当并发作业数目增加时，GraphCPP可以得到更好的性能提升。在作业数为2、4、8和16时，GraphCPP相对于SGraph-S的加速分别为xx，xx，xx和xx。这是因为随着作业数目的增加，GraphCPP通过摊销节省了更多的数据访问和存储开销。请注意，当只有一个并发作业时，GraphCPP的细粒度调度操作不会发生，因此此时和其他方案相比执行时间没有太大差异。根据我们的测试，GraphCPP上块调度成本只占总执行时间的xx-xx%。此外，由于对LLC等资源的争夺，并发版本（-C）的性能要比GraphCPP甚至顺序版本（-S）差得多。因此，简单地修改现有的图处理系统来支持并发任务可能是一个糟糕的选择。

然后我们评估了GraphCPP的横向扩展性。为了实现这一目标，我们首先在单个节点上增加CPU核数来评估GraphCPP在16个PPsP作业上的执行时间。从图17中可以看出，GraphCPP的性能在任何情况下都要好于其他方案，特别是在核数较多的情况下。这是因为GraphCPP的图数据是由并发任务共享的，而其他方案的数据访问成本更高。其次，我们还评估了1、2、4、8节点上不同方案的性能。如图18所示，GraphCPP在8个节点上的性能是单机的xx-xx倍，具有良好的扩展性。此外，GraphCPP的扩展性也要好于SGraph，PnP和Tripoline，这是因为数据共享和核心子图索引使得通信成本更低。因此我们相信GraphCPP可以高效地支持实际中的点对点查询应用。

相关工作

**点对点查询**。现有工作对点对点查询做出了许多研究，如𝐻𝑢𝑏2 [x]提出了一种以hub为中心的专用加速器，它认为具有大量连接的顶点，即hub，扩大了搜索空间，使最佳路径计算变得异常困难。它提出了hub-Network概念，以限制hub顶点的搜索范围。并使用hub2-Labeling方法来对hub搜索过程进行在线剪枝。但是由于𝐻𝑢𝑏2定位是专用加速器，它的通用性较差。PnP观察点对点查询的遍历过程，提出了基于上界的剪枝策略，减少了不必要的顶点遍历，为点对点查询的研究提供了新的思路。Tripoline通过日常维护一些“索引顶点”，以索引顶点为“中介”，推导两点之间近似的”上界“，这样实现了无需“先验知识”的上界查询。SGraph在前两者的基础上进一步发展，利用图上的三角不等式原理提出了基于上界和下界的剪枝策略，实现了亚秒级的图上点对点查询。但是这些系统都专注于优化单次点对点查询的速度，忽略了大规模并发查询的严重负载。

**并发图计算**。许多图计算系统都对并发计算进行了研究，GraphM指出并发图计算任务之间存在的“数据访问相似性”，并提出了一种以数据为中心的调度策略，实现多任务之间的数据共享，提高了并发图计算的吞吐量。但是GraphM是单机核外图计算系统，采用BSP计算模型，并且只适用于静态图。在此基础上，CGraph[x]进一步将应用场景扩展到分布式系统上的动态图计算，并针对分布式场景优化了通信机制和负载均衡策略，但是他和GraphM一样都是核外系统，即使可以通过调度策略将磁盘访问的开销分摊到不同子图，依然不适合并发查询的高负载场景。ForkGraph实现了在内存中进行高效地并发图处理，并且采用了基于让步的调度策略，每轮迭代仅处理部分数据，加速了整体执行速度。但是他是一个单机内存系统，并且没有为点对点查询进行优化，不适合在海量数据上执行并发点对点查询任务。

RELATED WORK

Point-to-Point Queries: Existing work has conducted extensive research on point-to-point queries. For instance, 𝐻𝑢𝑏2 [x] proposed a hub-centric specialized accelerator, which contends that vertices with a large number of connections, i.e., hubs, expand the search space, making best path calculations exceptionally challenging. It introduced the hub-Network concept to confine the search scope of hub nodes. The online pruning of hub search process was achieved using the hub2-Labeling method. However, due to 𝐻𝑢𝑏2's specialization in a dedicated accelerator, its applicability is limited. PnP observed the traversal process of point-to-point queries and introduced an upper-bound-based pruning strategy, reducing unnecessary vertex traversals and providing a fresh perspective for point-to-point query research. Tripoline derived an approximate "upper bound" between two points by maintaining some "permanent vertices" in daily operations, using them as intermediaries. This approach enabled "prior-knowledge-free" upper bound queries. SGraph further developed on the aforementioned methods, leveraging the triangle inequality principle on the graph to propose upper-bound and lower-bound pruning strategies, achieving sub-second point-to-point queries on the graph. However, these systems mainly focus on optimizing the speed of individual point-to-point queries, overlooking the severe load of large-scale concurrent queries.

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

结论

本文提出了一个并发点对点查询系统GraphCPP，它利用并发查询之间的数据相似性实现了多任务的数据共享。同时它采用轻量级的核心子图索引，更好地实现了多任务之间的计算共享。实验表明GraphCPP的性能优于目前最先进的图查询系统SGraph[x]、Tripoline[x]、Pnp[x] XXX倍。

致谢

CONCLUSION

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

ACKNOWLEDGMENTS

参考文献

1. Chen H, Zhang M, Yang K, et al. Achieving Sub-second Pairwise Query over Evolving Graphs[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 2023: 1-15.
2. Jiang X, Xu C, Yin X, et al. Tripoline: generalized incremental graph processing via graph triangle inequality[C]//Proceedings of the Sixteenth European Conference on Computer Systems. 2021: 17-32.
3. Xu C, Vora K, Gupta R. Pnp: Pruning and prediction for point-to-point iterative graph analytics[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 587-600.

废弃材料

废弃摘要内容：

在面对高并发的点对点查询需求时，由于冗余的数据访问，处理效率很低。我们观察到并发查询任务之间存在着数据访问相似性，这启发我们提出了一种以数据为中心的并发点对点查询方法。具体地，我们将图查询过程中的数据分为“图结构数据”和“任务特定数据”，前者记录了图的拓扑信息，后者记录了查询任务所要访问的图结构数据分块，不同查询独立访问任务所需的数据分块，这些分块可能重叠，但在传统的查询方案中。因此，我们采用了一种数据驱动的调度方法：在执行并发点对点查询任务时，内存/LLC中只保留一份图结构数据。多任务之间以细粒度的图数据分块为单位共享数据。一次访问，多个任务处理，以此分摊数据访问的开销，提高并发图查询的吞吐量。为了展示GraphCPP的效率，

核心子图查询机制

素材库：

CGP 作业固有的不规则访问导致由于局部性较差而导致底层内存子系统利用率不足。最终导致整个系统的吞吐量较低。首先，CGP作业由于其不同的遍历特性，对相同图结构数据表现出不规则的图遍历，并且这些作业同时访问同一图的不同部分。来自多个作业的这种不规则且不协调的内存访问会导致严重的缓存抖动。其次，CGP 作业对内存子系统造成激烈的资源争用。当在现有的多核处理器上运行多个作业时，这些作业会将与同一顶点关联的状态提取到不同的缓存行中。由于图的稀疏性，每个缓存行中只需要几个数据元素（甚至一个），因为图处理因展示对小数据元素（例如，每个顶点状态 4 或 8 字节）的固有随机访问而臭名昭著。整个图 [14,25,39]。

这会导致激烈的资源争用以及缓存和内存带宽的利用不足。

单作业加速器对于解决 CGP 作业之间不协调的图形数据访问效率低下

LCCG通过新的硬件机制增强了众核处理器：图遍历正则化和预取。前一个组件规范了 CGP 作业的图遍历，从根本上解决这些作业的不规则数据访问的挑战。与遍历正则化相结合，预取组件进一步隐藏了 CGP 作业的内存延迟，并有效地支持这些作业的合并访问。具体来说，顶点状态沿着图拓扑中固有的依赖关系传播。仅当其状态由其活动邻居更新时，非活动顶点才需要由作业处理。基于这一见解，提出了一种有效的拓扑感知执行方法，并得到 LCCG 的有效支持。它根据图拓扑动态探索所有 CGP 作业的以活动顶点为根的公共遍历路径，然后预取这些探索路径上的图数据，以驱动相应的作业一起同步处理这些数据。

CGP 作业会发出更多的冗余数据访问，并且由于不同作业在不同时间将更多冗余数据存储到缓存中，也会导致更严重的缓存干扰。它最终会导致系统吞吐量低下，因为数据访问成本通常占迭代图算法总执行时间 的主要部分。