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，在处理并发查询任务时，由于冗余的数据访问开销和计算开销，整体吞吐量很差。

本文提出了GraphCPP，它是第一个并发执行点对点查询任务的图遍历系统，通过数据访问共享和热路径计算共享提高并发查询任务的吞吐量。GraphCPP有两个创新之处。首先，基于不同查询任务的遍历路径大量重叠的观察，GraphCPP提出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； 其次，由于不同任务对相同的热门路径频繁计算，GraphCPP提出了一个基于核心子图的计算共享机制，它可以从已知的最佳路径中提取热路径，通过共享热路径的计算值来加速未知查询的收敛。为了展示GraphCPP的效率，我们在xx等数据集上，将其与最先进的点对点查询系统进行对比，例如SGraph、Tripoline、Pnp。结果表明，GraphCPP仅产生xx的预处理开销和xx的存储开销，但是整体吞吐量提高了xx-xx倍。

（无需翻译：参考了GraphM和OSDI22文章）

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 application of graph processing technology in fields such as map navigation and network analysis, there is a significant demand for high throughput in concurrent point-to-point query tasks on the same underlying graph. However, existing graph query systems have primarily focused on optimizing the speed of individual point-to-point queries. When dealing with concurrent query tasks, the overall throughput is compromised due to redundant data access overhead and computation expenses.

This paper introduces GraphCPP, the first graph traversal system designed for concurrent execution of point-to-point query tasks. It enhances the throughput of concurrent query tasks through data access sharing and hot path computation sharing. GraphCPP is novel in two ways. Firstly, based on the observation that traversal paths for different query tasks overlap significantly, GraphCPP proposes a data-driven caching execution mechanism. This mechanism, through fine-grained graph chunk scheduling, facilitates data sharing among concurrent tasks, thereby improving data access efficiency. Secondly, recognizing that different tasks frequently compute the same hot paths, GraphCPP introduces a computation sharing mechanism based on core subgraphs. This mechanism extracts hot paths from known optimal paths and accelerates the convergence of unknown queries by sharing computed values of these hot paths. To demonstrate the efficiency of GraphCPP, we compare it with state-of-the-art point-to-point query systems such as SGraph, Tripoline, and Pnp on datasets like xx. The results indicate that GraphCPP incurs only xx preprocessing overhead and xx storage overhead, achieving a throughput improvement of xx-xx times.

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

前言

大量并发点对点查询任务通常在同一底层图上执行，如：在物流运输时，谷歌地图[xx]会找到两个地点之间的最佳路径；在社交网络分析时，Facebook[x]会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，支付宝[x]会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[xxxx]聚焦于加速单次查询的效率，而忽略了对并发查询的优化。为了实现并发点对点查询任务的高效执行，需要解决两个关键的挑战。

首先，当在同一个底层图上执行并发点对点查询任务而不考虑数据访问相似性，会导致对重叠数据的冗余访问。具体来说，不同查询查询任务从不同的起点出发，最终到达各自的终点。它们的遍历路径存在着大量重叠，然而由于不同查询任务访问数据的重叠部分各不相同，且它们会沿着不同的路径访问重叠图数据。因此现有的查询系统采用保守的策略，让每个任务负责访问各自所需的数据。这意味着每个任务的数据访问是完全独立的，即使他们的遍历路径高度重合，如一对查询的路径是另一对查询的路径的子集，依然需要重复加载重叠部分的数据，无法享受重用缓存中数据的好处。

（无需打印：<https://www.alibabagroup.com/document-1595215205757878272>

参考graphfly）

其次，除了重复访问外，不同查询任务还需要重复计算热门路径的距离值。由于图数据往往具有幂律分布的特点，少量热门顶点会频繁出现在不同查询的最佳路径中。由于不同查询任务之间不会共享计算，它们会重复计算连接热顶点的热路径。此外，热门顶点往往是具有大量邻居顶点的高度顶点，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[xxxx]的方式来实现计算共享，但由于昂贵的计算开销、存储开销、更新开销限制了计算共享的效率和精确度。

A significant number of concurrent point-to-point query tasks are commonly executed on the same underlying graph. For instance, in logistics and transportation, Google Maps [xx] identifies the optimal route between two locations. In social network analysis, Facebook [x] recommends potential friends to users by exploring the relationship chain between two users. In financial risk analysis, Alipay [x] analyzes how risk spreads from one entity to another. These popular applications highlight the demand for executing large-scale concurrent point-to-point queries on the same underlying graph. However, existing solutions for point-to-point queries [xxxx] primarily focus on accelerating the efficiency of individual queries while neglecting optimization for concurrent queries. To achieve the efficient execution of concurrent point-to-point query tasks, two key challenges need to be addressed.

Firstly, executing concurrent point-to-point query tasks on the same underlying graph without considering data access similarity leads to redundant access to overlapping data. Specifically, different query tasks start from distinct origins and eventually reach their respective destinations. There is a significant overlap in their traversal paths. However, as the overlapping portions of data accessed by different query tasks vary, and they traverse overlapping graph data along different paths. Therefore, existing query systems adopt a conservative strategy, assigning each task to access the data it requires independently. This means that the data access for each task is entirely isolated, even if their traversal paths highly overlap. For example, if the path of one query is a subset of the path of another query, the overlapping data needs to be loaded redundantly, preventing the benefits of reusing data in the cache.

Furthermore, in addition to redundant access, different query tasks also face the challenge of recalculating distance values for popular paths. As graph data often exhibits a power-law distribution, a small number of popular vertices frequently appear in the optimal paths of different queries. Due to the lack of computation sharing among different query tasks, they redundantly compute the hot paths connecting popular vertices. Moreover, popular vertices often correspond to high-degree vertices with numerous neighboring vertices, and the repetitive traversal of these vertices results in an explosive growth in computation costs. Some existing systems have attempted to implement computation sharing through global indexing [xxxx], but the efficiency and accuracy of computation sharing are limited by the expensive computational, storage, and update costs.为了应对上述挑战，本文提出了一个数据驱动的并发点对点查询系统，称为GraphCPP。它通过数据共享机制和计算共享机制，提高了并发点对点查询系统的吞吐量。

在GraphCPP中，我们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，关联任务数量越多的分块优先级越高更可能被优先调度。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；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。在按照优先级顺序将图分块加载到LLC后，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，GraphCPP提出了一个基于核心子图的查询加速机制，以实现并发图查询任务的计算共享。它将图中具有大量连接边的顶点称为热顶点，它的作用类似于现实世界中航空网络中作为中介的枢纽城市。将热顶点之间的路径称为热路径，虽然数量较少，但是在不同查询任务到的遍历路径中频繁出现。构建核心子图首先要从原始图数据中筛选高度顶点，接着遍历这些顶点之间的热路径，将热路径的路径值作为热顶点对的边的权重。在查询过程中，核心子图就像一个高速公路网，查询任务的起点到达一个热顶点（相当于从起点进入高速公路的入口），然后前往另一个热顶点（相当于经由高速公路到达出口），最后到达目的顶点（相当于从高速公路出口到达目的点）。GraphCPP通过预计算获得热路径的值，并保存在核心子图中，从而实现计算共享。

最后，GraphCPP还通过预测不同查询任务的遍历路径，驱动高度重叠的相似查询任务批量执行，进一步提高了并发查询的性能。

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

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

Secondly, GraphCPP proposes a query acceleration mechanism based on core subgraphs to achieve computation sharing in concurrent graph query tasks. It identifies vertices with a significant number of connecting edges in the graph as hot vertices, which play a role similar to hub cities acting as intermediaries in real-world airline networks. The paths between hot vertices are referred to as hot paths, and although their quantity is limited, they frequently appear in traversal paths reached by different query tasks. Constructing a core subgraph involves first selecting high-degree vertices from the original graph data. Subsequently, it traverses the hot paths between these vertices, assigning the path values as weights to the edges between pairs of hot vertices. During the query process, the core subgraph acts as a high-speed network of highways. The query task starts at a hot vertex (analogous to entering a highway at an entrance), moves to another hot vertex (similar to reaching an exit via the highway), and finally arrives at the destination vertex (analogous to reaching the destination from a highway exit). GraphCPP achieves computation sharing by precomputing the values of hot paths and storing them in the core subgraph.

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

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和计算相似性来提高并发任务吞吐量。
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:

We analyze the performance bottlenecks of existing point-to-point query systems when handling concurrent tasks and propose leveraging the similarities in data access and computation among concurrent tasks to enhance throughput.

We develop GraphCPP, a dynamic graph-driven concurrent processing system for point-to-point queries. It achieves data sharing and computation sharing among concurrent tasks and introduces a strategy for batch execution of similar tasks.

We compare GraphCPP with the state-of-the-art point-to-point query system XXXXXX. The results indicate XXXXXXXX.

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。

大多数现有的解决方案[xxxx]聚焦于加速单次查询的速度，然而实际应用场景中，有大量图查询任务在同一个底层图上并发运行。例如，中国地理信息产业协会的统计]表明，百度地图[xx]、高德地图[xx]、腾讯地图[xx]、华为地图[xx]等企业建设的位置服务开放平台日均位置服务请求次数最高达1600亿次[xx]。大量并发点对点查询需求对图遍历系统的吞吐量提出了极高的要求。然而如图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和冗余计算。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

Most existing solutions [xxx] are primarily focused on accelerating the speed of individual queries. However, in practical scenarios, there is a significant number of graph query tasks concurrently running on the same underlying graph. For instance, statistics from the China Geographical Information Industry Association indicate that location service open platforms constructed by companies such as Baidu Maps, Gaode Maps, Tencent Maps, and Huawei Maps receive a daily average of up to 160 billion location service requests. The substantial demand for concurrent point-to-point queries poses a high requirement for the throughput of graph traversal systems. Yet, as illustrated in Figure 2 (showing the parallel query execution time for different systems, highlighting poor parallel execution efficiency), we demonstrate that existing systems exhibit low throughput when handling large-scale concurrent queries. The root cause of this undesirable outcome is the significant redundancy in data access and computation among concurrent tasks. To qualitatively analyze the aforementioned issues, we conducted performance evaluations on parallel point-to-point queries using XXXXX (machine configuration), selecting XXXXX (the current best solution) on XXXXX (graph dataset).

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：（界限）主流的点对点查询系统中，普遍采用了基于剪枝的查询策略， 界限提供了保守的剪枝值。具体地，界限可进一步划分为上界和下界，上界UB表示当前已知的从源点到目的顶点的最佳路径的路径值，下界则LB表示从当前顶点v到目的顶点保守的最佳距离预测值，预测的LB小于或等于顶点v到目的顶点实际的最佳距离。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。

定义五：（核心子图）我们使用Gcore=(Vhot,Ehot,Indexhot)来表示核心子图，其中Vhot是热顶点（具有较多连接边）的集合，Ehot将Vhot中热顶点之间的热路径抽象为一条连边,Indexhot表示Ehot对应的热路径的路径值。

**Preliminaries**

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

Definition 2: (Graph Partition) We use Pi=(VPi, EPi) to represent the i-th graph partition of a directed graph. VPi denotes the set of vertices in the graph partition, and EPi is the set of directed edges formed by vertices in VPi. For distributed systems, different machines have distinct graph partitions Pi. We adopt edge-cut partitioning, where the same vertex may appear on different computation nodes, but only one is the primary vertex, and others are mirrored vertices.

Definition 3: (Point-to-Point Query) We use qi=(si, di) to denote the query corresponding to task i, where si and di represent the source and destination vertices of query qi, respectively. The result value Rsd obtained by query qi has different meanings for different algorithms; for example, in the case of the Shortest Optimal Path query, Rib represents the shortest optimal path between si and di. We use Q={q1, q2, … q|Q|} to represent the set of concurrent point-to-point queries, where |Q| indicates the total number of queries.

Definition 4: (Bounds) In mainstream point-to-point query systems, a pruning-based query strategy is widely adopted, where bounds provide conservative pruning values. Specifically, bounds can be further categorized into upper bounds (UB) and lower bounds (LB). UB represents the current known optimal path value from the source to the destination vertex, while LB signifies a conservative predicted distance value from the current vertex v to the destination vertex, with the predicted LB being less than or equal to the actual optimal distance from vertex v to the destination vertex. Following the triangle inequality on the graph, if the distance of a path is greater than UB or exceeds UB when adding the value of LB, the path is definitely worse than existing paths and needs to be pruned. The values of upper and lower bounds need indexing to be derived, essentially constituting a form of computation sharing.

Definition 5: (Core Subgraph) We use Gcore=(Vhot,Ehot,Indexhot) to represent the core subgraph, where Vhot is the set of hot vertices (with many connecting edges), Ehot abstracts the hot paths between vertices in Vhot as edges, and Indexhot represents the path values corresponding to Ehot.

Definition 6: (Index) The index records the optimal query path values between vertex pairs. GraphCPP sorts vertices by degree, selecting k+m vertices with the highest degrees (values for k and m are generally user-determined, with k usually set to 16, and m typically one order of magnitude larger than k). The first k vertices serve as global index vertices, while the remaining vertices function as core subgraph index vertices. The global index records optimal query path values to all vertices in the graph. The core subgraph index only records query path values between hot vertices in the subgraph.

定义六：（索引）索引记录了一个顶点对之间的最佳查询路径值。GraphCPP将顶点按照度数排序，选择k+m个度数最高的热顶点（k和m的值一般由用户确定，k值一般取16，m一般比k大一个数量级伊桑），其中前k个顶点为全局索引顶点，剩下的顶点作为核心子图索引顶点。全局索引记录了到达图中所有顶点的最佳查询路径的路径值。核心子图索引只记录子图中热顶点之间的查询路径值，

并发点对点查询任务的性能瓶颈

在本节，我们在gemini系统的基础上实现了当前最先进的点对点查询系统SGraph的并发版本SGraph-C，并通过 Twitter评估了SGraph-C执行并发点对点查询的性能。以定量地说明现有并发点对点查询方案的性能瓶颈。

**并发任务的****冗余数据访问开销。**我们观察到当并发点对点查询任务在同一个底层图上执行图遍历时，体现出来很明显的数据访问相似性。如图x，随着并发查询任务数的提高，有越来越多的图数据分块被1个以上的任务共享。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。这意味着需要在资源有限的缓存中保存相同图分块的多个副本，从而加剧了缓存冲突。图x表明，随着并发数目的增大，查询任务的缓存未命中率也急剧增大。如图x所示，虽然并发执行的整体效果优于线性执行，但是由于前面提到的冗余数据访问开销，随着查询任务数量额增加每个任务的平均查询时间显著增加。

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

In this section, we have developed the concurrent version SGraph-C based on the Gemini system, which is currently at the forefront of pairwise query systems. We evaluated the performance of SGraph-C in executing concurrent pairwise queries through Twitter, aiming to quantitatively illustrate the performance bottlenecks of existing concurrent pairwise query solutions.  
The redundant data access overhead of concurrent tasks is evident. We observe that when concurrent pairwise query tasks execute graph traversal on the same underlying graph, a significant portion of their traversal paths exhibits clear data access similarities. As shown in Figure 3x, our data indicates that... with the increase in the number of concurrent query tasks, more and more graph data chunks are shared by more than one task. However, in the traditional "task->data" scheduling mode, different tasks independently load the graph data chunks they need into the cache for processing at different times. This means that multiple copies of the same graph chunk need to be stored in the cache, exacerbating cache conflicts. Due to the limited space of the cache, Figure x illustrates that, as the concurrency increases, resource competition between query tasks leads to a significant increase in cache miss rates, resulting in substantial performance loss. As shown in Figure x, although the overall performance of concurrent execution is better than linear execution, the average query time per task significantly increases with the number of query tasks due to the mentioned redundant data access overhead.

**并发任务的冗余计算开销。**由于图数据的幂律分布特性，少数热顶点连接了大部分边。因此，如图x所示，尽管热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%）。并且拥有更高共享程度的图分区热顶点的比例越高。这意味着不同的查询任务会重复计算热顶点之间的热路径，体现出了并发任务之间的计算相似性。在一个图快照周期内，对相同顶点对的路径计算结果是相同的，这意味着并发任务的大量计算是冗余的。此外，如图x所示，由于热顶点往往拥有大量出边和入边，它带来的计算开销也远大于普通的顶点。

一些现有的解决方案尝试建立全局索引机制[x]来利用计算相似性。然而全局索引机制存在天然缺陷。缺陷1：全局索引需要记录高度顶点与其它所有顶点的路径值，当图的规模非常大时，建立索引的计算开销和存储开销会很大。如图x和x所示，全局索引的计算开销和存储开销，随着全局索引顶点数的增加而等比例增加。缺陷2：在流图上的点对点查询中，每轮图更新都会有新的边添加和边删除产生，全局索引需要基于最新的图快照来进行动态更新高度顶点与每一个顶点的索引关系，这意味着流图的任何更新都会对所有的顶点索引造成影响。如图x所示，少量图更新会导致全局索引的大规模更新。缺陷3：不同图数据集的数据分布情况也不同，很难选择一个合适的索引顶点数目以在计算共享的有效率和索引本身的开销之间取得平衡。如图x所示，在不同的数据集上选取同样的索引顶点数，其固有开销和计算共享的有效性变化很大。综上，全局索引机制本身会带来昂贵地开销，且针对不同的数据集所需开销的波动很大。因此现有的系统往往保守地选取全局索引的数目（在Tripoline和SGraph中都选择16个顶点），以避免带来昂贵地开销，但这样也限制了全局索引的覆盖范围，无法实现高效地计算共享。

Redundant Computational Overhead of Concurrent Tasks. Due to the power-law distribution characteristics of graph data, a small number of hot vertices are connected to the majority of edges. Therefore, as shown in Figure X, although hot vertices constitute only a small portion of the total vertices (XX%), they appear in many paths (XX%). The proportion of hot vertices in graph partitions with higher sharing levels is even higher. This implies that different query tasks will redundantly calculate hot paths between hot vertices, demonstrating computational similarity among concurrent tasks. Within a graph snapshot period, the results of path calculations for the same vertex pairs are identical, indicating that a significant amount of computation in concurrent tasks is redundant. Additionally, as illustrated in Figure X, due to the fact that hot vertices often have a large number of outgoing and incoming edges, the computational cost they bring is much greater than that of ordinary vertices.

Some existing solutions attempt to establish a global indexing mechanism [X] to leverage computational similarity. However, the global indexing mechanism has inherent flaws. Flaw 1: Global indexing requires recording the path values between high-degree vertices and all other vertices. When the graph scale is extremely large, the computational and storage costs of building the index can be substantial, as shown in Figures X and X, where the computational and storage costs of global indexing increase proportionally with the number of global index vertices. Flaw 2: In point-to-point queries on streaming graphs, each round of graph updates introduces new edges and edge deletions. The global index needs to dynamically update 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 will impact all vertex indices, as illustrated in Figure X, where a small number of graph updates lead to extensive updates in the global index. Flaw 3: Different datasets have different data distribution patterns, making it challenging to choose an appropriate number of index vertices to balance the efficiency of computational sharing and the overhead of the index itself. As shown in Figure X, selecting the same number of index vertices on different datasets results in significant variations in inherent costs and the effectiveness of computational sharing.

In summary, the global indexing mechanism itself incurs expensive costs, and the fluctuation in required costs for different datasets is considerable. Therefore, existing systems often conservatively choose the number of global indices (e.g., 16 vertices in Tripoline and SGraph) to avoid incurring excessive costs. However, this also limits the coverage range of global indices, preventing efficient computational sharing.

我们的启发

通过上述的观察，我们得到了以下启发：

**启发**1：不同查询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，对重叠数据的访问产生了冗余开销。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。

**启发2**：不同查询任务之间存在计算相似性。由热顶点和热路径组成的核心子图就像是道路交通网络中的高速公路网，会频繁地被不同的任务重复遍历。但是现有的图遍历系统要么没有利用到计算相似性[xx pnp]，要么采用代价昂贵的全局索引机制[xx Tripoline]，限制了计算共享的效果。这启发我们通过建立轻量级地核心子图机制，在不同查询任务中共享热路径计算结果。

**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；GraphCPP Overview

To enhance the efficiency of concurrent point-to-point queries, we introduce a data-driven efficient concurrent point-to-point query system, GraphCPP, after a thorough investigation into the computational details of concurrent point-to-point queries. The key idea of our system is to achieve data and computation sharing among concurrent point-to-point query tasks. As illustrated in the figure below, GraphCPP proposes an efficient data-driven caching execution mechanism, utilizing data similarity among concurrent tasks to enable shared access to overlapping data. It also introduces a query acceleration mechanism based on the core subgraph, facilitating the sharing of computations for the same hot paths across different query tasks. Additionally, it leverages path prediction for different queries, driving the bulk execution of similar queries with overlapping paths, further exploiting data similarity.

{\bf{Data Access Sharing Mechanism}}.Responsible for fine-grained sharing of graph structure data utilizing the data access similarity among concurrent tasks. Initially, it partitions the original graph data into coarse-grained graph partitions, distributed for parallel processing across different machines, similar to other distributed graph computing systems [xx]. Subsequently, a fine-grained chunk manager is employed to further divide coarse-grained graph partitions into fine-grained graph chunks. The association mechanism between graph chunks and tasks is then executed: if a query task qi has active vertices in a graph chunk bi, a relationship between qi and bi is established. The chunk priority scheduling mechanism prioritizes graph chunks with more associated tasks to be loaded into the Last Level Cache (LLC). The associated task trigger, based on active graph chunk information in LLC and chunk-task association information, selects all tasks with an association to execute in batch on shared graph chunks.

{\bf{Calculation Sharing Mechanism}}. It is responsible for leveraging index information (including global index and core subgraph index) to utilize computational similarity between concurrent query tasks for shared path computation. 1) Preprocessing Phase: It first gathers degree information for all vertices in the original graph data, sorts the vertices in descending order based on their degrees, and selects vertices ranked from 1 to k as global index vertices and vertices ranked from k+1 to k+m as core subgraph index vertices. 2) Index Building Phase: On one hand, before queries start, it executes a point-to-multipoint query algorithm to compute the best path values from global index vertices to all vertices in the graph (e.g., for PPSP tasks, it requires executing SSSP algorithm). For directed graphs, it calculates both the outbound and inbound path values for global index vertices, while for undirected graphs, only one calculation is needed. On the other hand, GraphCPP dynamically maintains the core subgraph through a runtime method. Specifically, any segment of a best path between a vertex pair corresponds to the best path between the corresponding vertices. Therefore, the core subgraph does not need precomputation for hot paths but dynamically explores hot paths after each query, adding them to the core subgraph structure. 3) Computation Sharing Phase: Global index vertices typically act as intermediaries for numerous paths, so at the beginning of a query, using global index vertices can compute the path value for a reachable path. This path may not necessarily be the best path between query vertex pairs but can provide a reference for pruning queries. Moreover, in pruning traversal based on upper bound + lower bound, global index can estimate the path value for query paths earlier, allowing earlier pruning of unsuitable paths. Using global index constitutes the first level of computation sharing. The core subgraph maintains best path values between hot vertices, serving as a highway network between query tasks. When a query task traverses hot vertices in the core subgraph, it connects to the highway. Leveraging the core subgraph, it can quickly reach the exit vertex from the entrance vertex without recalculating the path value for this hot path segment. Using the core subgraph constitutes the second level of computation sharing.

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。

整体执行流程

算法x展示了GraphCPP处理并发查询任务的整体执行流程。假设我们已经完成分区，在第一行我们得到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，并为他们分配一段内存空间。在第二行，我们进入一个循环处理过程，查询会迭代进行直至收敛。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 |
| 1: function OverallWorkflow  2 MallocBuffers( *B*, *Q* ) //*B* is the set of graph blocks, and *Q* is the set of query tasks  3: While has\_active( *B* ) do：  4: *bi* ← ChoseNextSharingBlock( )  5: *Qbi* ← ChoseAssociatedQueries( *bi* )  6: Parallel\_for\_each *qi* ∈ *Qbi* do: // Execute queries in *Q* in parallel, which is associated with block *bi*  7: new\_query =GraphCPPCompute( *qi*, *bi* ) // The implementation function for point-to-point queries returns the active vertex set after one round of task iteration.  8: if(has Associated( ( *bi* , new\_query ) ):  9: *Qbi*.Push( new\_query )  10: else:  11: *Q*.Push( new\_query ) |

Algorithm x illustrates the overall execution process of GraphCPP in handling concurrent query tasks. Assuming partitioning is complete, in the first line, we obtain the set B, which includes all graph blocks on the current computing node, and the set Q, which includes all query tasks on the current computing node. Each of them is allocated a segment of memory. In the second line, we enter a loop processing phase, and queries iterate until convergence is achieved. GraphCPP calls ChooseNextSharingBlock to update the association between query tasks and graph blocks and selects the graph block b\_i with the highest priority (having the most associated tasks). By tallying the associated blocks for each task (i.e., tasks with active vertices in the current block), we can determine all query tasks related to the current graph block b\_i (fourth line). Next, we load b\_i into the cache and parallelly process all associated query operations q\_i (fifth line). We invoke GraphCPPCompute to perform point-to-point query operations q\_i on the current block. If the query is not finished, we update the state of query q\_i, generating new query tasks (sixth line). If the newly generated query is associated with the current graph block b\_i, we add q\_i to Q\_b\_i, then return to the fifth line to continue the query. Otherwise, we save the information of the newly generated query to the query task set, and the task is suspended.



上述算法展示了GraphCPP中的整体工作流程，其在下面的章节，我们将详细介绍数据访问共享和计算共享两个优化机制。

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

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. To implement this execution model, we need to address two issues: 1) How to determine the shared data segments? 2) How to implement data sharing among multiple tasks? Below are our implementation details.

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, 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。代码x展示了GraphCPP划分图分块的过程：

|  |
| --- |
| 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（第一行）。在第二行，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。第四行，GraphCPP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(第6行)。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（第8行）。每次遍历完一条边都会判断当前分块是否已满（第11行），若分块已满，将当前分块加入block\_set（第12行），并清空记录的分块信息（第13行）。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。

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

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

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

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

三、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和计算相似性。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。

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 distance values from high-degree vertices to other vertices, thus enabling the sharing of these high-degree vertex distance values across different query tasks. However, the global index mechanism exhibits the following shortcomings: Shortcoming 1: The global index necessitates the recording of distance 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 distance 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: Core Subgraph Query Algorithm |
| 1: global\_index = BuildGlobalIndex(k) // Step 1: Calculate k Global Index  2: core\_subgraph\_index = BuildCoreSubgraphIndex(m, global\_index) // Build m Core Subgraph Index  3: function GraphCPPCompute(q, b)  4: active\_vertices = InitializeActiveVertices(q, b) // Determine active vertices based on query task and graph block  5: InitializeBoundsFromGlobalIndex(global\_index) // Initialize bounds based on the global index  6: while active\_vertices is not empty:  7: for vertex in active\_vertices:  8: if vertex is in core\_subgraph:  9: UpdateBoundsByCoreVertices(vertex, core\_subgraph\_index)  10: else:  11: for nbr in GetOutgoingNeighbors(vertex): // Traverse outgoing neighbors of vertex  12: UpdateBoundsByNeighbors(nbr)  13: end for  14: end for  15: active\_vertices = UpdateActiveVertices() |

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 shortest paths (including distance 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 shortest 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 distance 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 shortest 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 shortest 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。

二、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，无序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。

实验评估

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

实验设置

预处理开销

整体性能对比

调度策略性能

是否开启索引子图对结果影响

可扩展性

2. Batch Execution of Similar Tasks

Different query tasks arrive randomly, and they often follow distinct traversal paths. We observed that when two tasks have very low similarity, their overlapping path proportion decreases, and there might be no overlap at all. However, if the starting and target vertices of two queries are located in adjacent data blocks within the graph, the paths they traverse during the query process are highly likely to be close to each other. To address this, we propose a strategy for batch executing similar tasks, selecting batches of similar tasks from the task pool at a time to further leverage 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 executes a k-hop SSSP to retrieve the neighboring vertex sets SetS for the starting vertex and SetD for the target vertex (the value of k is determined by the user and is typically set to 3 by default). Subsequently, it iterates through the task pool, filtering out all the query tasks where the starting point is in SetS and the target point is in SetD. These tasks are treated as similar tasks and executed concurrently.It is worth noting that if the starting or target vertex of a query belongs to a high-degree vertex, the index can be directly used to accelerate the query process, bypassing the regular query steps. Excluding the high-degree vertices, the overhead of the k-hop SSSP is minimal, and the execution can be done concurrently with normal queries, with negligible additional costs.

EXPERIMENTAL EVALUATION

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 shortest 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.
4. <https://www.alibabagroup.com/document-1595215205757878272>
5. [www.google.com/maps](http://www.google.com/maps)
6. [www.facebook.com](http://www.facebook.com)
7. [www.alipay.com](http://www.alipay.com)
8. <http://www.cagis.org.cn/>
9. [www.baike.baidu.com](http://www.baike.baidu.com)
10. [www.ditu.amap.com](http://www.ditu.amap.com)
11. [www.http://map.qq.com/](http://www.http://map.qq.com/)
12. [www.petalmaps.com](http://www.petalmaps.com)
13. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning
14. Efficient Disk-Based Directed Graph Processing: A Strongly Connected Component Approach
15. Gemini: A Computation-Centric Distributed Graph Processing System

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

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；

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。

中心网络可以被认为是由中心锚定的高速公路结构，用于在大型社交网络中路由最佳路径。由于枢纽的重要性，非枢纽顶点对之间的大多数最佳路径可能需要经过这样的网络，即起始顶点到达一个枢纽（作为高速公路入口），然后前往另一个枢纽（作为高速公路入口）出口），最后离开高速公路到达目的地。换句话说，集线器网络可用于限制（或优先考虑）集线器的邻居；集线器只能在集线器网络内扩展。