目录

[动态图上高效缓存的并发处理点对点查询策略 2](#_Toc146465686)

[摘要 2](#_Toc146465687)

[前言 4](#_Toc146465688)

[背景和动机 6](#_Toc146465689)

[系统概述 6](#_Toc146465690)

[基于xxxxx的优化策略 6](#_Toc146465691)

[实验评估 6](#_Toc146465692)

[相关工作 6](#_Toc146465693)

[结论 6](#_Toc146465694)

[参考文献 7](#_Toc146465695)

动态图上以数据为中心的并发处理点对点查询系统

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，对现有的图查询系统提出了严重挑战。针对并发点对点查询有两个优化思路，1，加快对单个查询的响应速度；2，通过数据共享优化并行查询任务的数据访问效率；对于前者，我们提出了一个高速地核心子图查询机制，根据顶点度数筛选出两级常设顶点，所有常设顶点组成核心子图，通过特定的规则维护核心子图上的索引值，借助它可以在查询到来时快速确定上界近似值，从而加速单次查询；对于后者，我们提出了一个以数据为中心的缓存处理机制，它将图数据结构划分为LLC级别的分块，并通过优先级调度策略和细粒度同步策略实现多任务之间数据共享，优化缓存命中率，提高并发查询的吞吐量。据我们所知，GraphCPP是第一个针对并发点对点查询场景进行优化的工作，我们将其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。

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

Abstract

With the widespread application of graph processing techniques in areas such as map navigation and network analysis, a large number of point-to-point query tasks concurrently operate on the same underlying graph, posing serious challenges to existing graph query systems. This paper proposes two optimization strategies for concurrent point-to-point queries: 1) accelerating the response time for individual queries; 2) optimizing data access efficiency for parallel queries through data sharing. For the former, we introduce a high-speed core subgraph query mechanism (Substrate）that maintains a core subgraph composed of highly connected vertices to quickly determine upper-bound approximations upon query arrival, thereby accelerating single queries. For the latter, we propose a data-centric processing substrate that partitions the graph data structure into LLC-level blocks. By employing priority scheduling and fine-grained synchronization strategies, it facilitates data sharing among multiple tasks, enhancing the throughput of concurrent queries. To the best of our knowledge, GraphCPP is the first work optimized for concurrent point-to-point query scenarios. Comparative experiments with state-of-the-art point-to-point query systems, including SGraph[x], Tripoline[x], and Pnp[x], demonstrate that GraphCPP improves the efficiency of concurrent point-to-point queries by a factor of xxxx.

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节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。利用专门设计的算法，如Point-to-Point Shortest Path (PPSP)、Point-to-Point Widest Path (PPWP) 以及 Point-to-Point Narrowest Path (PPNP)，可以在无需（并不能完全避免）查询与处理不相关的其他节点或边的情况下，精确地确定两节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短路径；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。

通常来说，要提高图上并发点对点查询的吞吐量，可以从两方面入手：1，加快单次查询的速度；2，采用高效地调度策略优化并行查询的效率；当前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，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通过维护中心节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。然而我们发现上述工作都没有考虑高并发的点对点查询场景。我们在2.2节证明了在企业应用中经常要面临高并发的查询需求，而现有系统存在冗余的数据访问开销，常常造成严重的性能瓶颈。为此，本文提出了GraphCPP，一种以数据为中心的并发处理点对点查询系统，它对单次查询和并发查询都做了专门优化。

INTRODUCTION

Point-to-point query tasks on graphs refer to the exploration of specific relationships between two distinct objects using the graph as a generic data structure. Unlike traditional graph query methods, point-to-point queries focus exclusively on the analysis of associations or paths between two specific nodes, without the need for complex queries involving the entire graph or its large-scale subsets. This targeted query strategy endows point-to-point queries with significant optimization potential. Utilizing specially designed algorithms such as Point-to-Point Shortest Path (PPSP), Point-to-Point Widest Path (PPWP), and Point-to-Point Narrowest Path (PPNP), specific path attributes between two nodes can be accurately determined without the need to query and process unrelated nodes or edges. Due to the efficiency of point-to-point queries in graph analysis, they have found widespread practical applications in various fields. For example, in logistics, finding the shortest path between two locations（举例子xxxxx）; in social network analysis, recommending potential friends by examining the relationship chain between two users（举例子xxxxx）; in financial risk analysis, analyzing how risk propagates from one entity to another（举例子xxxxx）. These popular applications have created a demand for executing large-scale concurrent point-to-point queries on the same underlying graph.

To enhance the throughput of concurrent point-to-point queries on graphs, two approaches can be taken: 1) speeding up the response time for individual queries; 2) employing efficient scheduling strategies to optimize parallel query efficiency. Existing solutions for point-to-point queries primarily focus on accelerating the efficiency of individual queries.

**单次点对点查询优化**：GraphCPP提出了一个高速地核心子图查询机制，它的核心思想是利用闲时算力维护常设顶点之间的索引，加速突发查询请求的计算过程。具体地，我们首先观察到大部分点对点查询的路径会在少部分特定路段上发生重叠，而这部分特定路段通常经过高度顶点。我们把这部分高度顶点称为常设顶点，把常设顶点组成的子图称为核心子图。我们为每一个常设顶点维护一个索引值，它记录了常设顶点与其它所有顶点的距离值。当有新的点对点查询需求到来时，首先查阅核心子图，从索引中发现一条源点到目的顶点的最短路径。由于最终的路径不一定经过核心子图上的顶点，这条路径不一定是全局最短的，但是却可以为我们提供一个比较精确的上界值，利用它，可以对点对点查询过程进行剪枝，大大加快查询速度。此外，点对点查询在双向搜索的过程中，还可以利用核心子图更新上界值，优化剪枝的精度。具体见第四章节。

**并发查询优化**：GraphCPP提出了一个以数据为中心的缓存处理机制，核心思想是改变并发任务的调度顺序，通过细粒度调度顺序来实现多任务之间重叠数据访问的共享，具体包含以下几部分：**1，确定数据共享部分**：在GraphCPP框架内，我们对每个计算节点的图结构数据进行了更为细粒度的划分，使其适配LLC的大小。接着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区优先级越高，更可能被优先缓存，从而加速计算并提升整体效率；**2，实现多任务间数据共享**：传统策略中任务的数据访问彼此独立，即使它们处理的数据完全相同也无法共享。GraphCPP提出了一种关联任务触发机制，根据加载到LLC中的图分块内容，触发有关联得查询任务批量执行。由于多任务访问相同数据可能导致冲突，GraphCPP提出了细粒度的同步策略，以非原子操作的方式实现了对共享数据的高效访问，提高并发任务的吞吐量。For instance, Pnp employs a lower-bound-based pruning method to reduce redundant accesses during the query process. Tripoline maintains daily indices from central nodes to other vertices, enabling rapid queries without prior knowledge. SGraph leverages the triangle inequality principle, proposing an "upper-bound + lower-bound" pruning method to further reduce redundant accesses during point-to-point queries. However, none of the aforementioned works consider high-concurrency point-to-point query scenarios. In Section xxx, we demonstrate the frequent need for high-concurrency queries in enterprise applications, and highlight the presence of redundant data access costs in existing systems, often leading to severe performance bottlenecks. To address this, this paper introduces GraphCPP, a data-centric concurrent point-to-point query system that is optimized for both single and concurrent queries.

**Single Point-to-Point Query Optimization:** GraphCPP introduces a high-speed core subgraph query mechanism. Its execution steps are as follows: Index Construction: It first traverses all partitions, counts vertex degrees, and selects Hub vertices. Hub vertices are selected based on at least one of the following two criteria: 1) the vertex is among the top k vertices in the complete graph by degree; 2) the vertex is among the top q vertices in its respective partition by degree (the sizes of k and q need to be adjusted based on the scale of the graph and memory capacity). Once Hub vertices are determined, GraphCPP performs Single-Source Shortest Path (SSSP) from each Hub vertex on the entire graph, recording the distance values from the Hub vertex to all other vertices (for directed graphs, it also records the distance values from all other vertices to the Hub vertex), which we refer to as the Hub vertex's index value. Query Execution: The index value of Hub vertices allows us to quickly find the safe approximate value for the corresponding query.

综上，本文主要做出了如下贡献：

1. 揭示了现有图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。并指出可以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. 实现了GraphCPP，一个动态图上以数据为中心的并发处理点对点查询系统，它利用核心子图机制优化单次查询速度。然后利用并发任务之间的数据访问相似性，加速并发点对点查询系统的吞吐量。
3. 我们将GraphCPP于当前最先进的点对点查询系统XXXXXX进行对比，结果表明XXXXXXXXX

Specifically, when a query arrives, all Hub vertices, along with the source and destination nodes, form a core subgraph. Since statistics have already been completed during index construction, the weights of all edges on the subgraph are known. Point-to-point queries are performed on the source and destination vertices of the core subgraph, resulting in a distance value, w. This value may not be the shortest path value, but it provides a reference for our traversal and can be used for pruning. Pruning Query: Similar to the operation of SGraph, GraphCPP uses the obtained value w as the pruning "upper bound," pruning all paths with distance values greater than this upper bound. Additionally, GraphCPP leverages index values, w, and the triangle inequality to deduce the lower bound of the path. All paths with values greater than w, considering the lower bound, are also pruned.

**Concurrent Query Optimization:**GraphCPP introduces a data-centric processing substrate, consisting of the following components: 1) Priority-based Graph Partition Scheduling Strategy: Within the GraphCPP framework, we partition the graph structure data of each computing node into finer-grained blocks to fit the LLC size. Furthermore, we propose a priority calculation method. This method aims to closely associate query tasks with their relevant graph partitions. An increase in the number of tasks associated with a partition leads to an elevation in its priority, making it more likely to be prioritized for caching, thereby accelerating computation and improving overall efficiency. 2) Concession-based Asynchronous Task Execution Strategy: Traditional strategies tend to ensure the rapid completion of a single task, which may lead to some tasks finishing prematurely, while others wait for an extended period, constraining the throughput of large-scale concurrent queries. GraphCPP introduces a concession-based asynchronous strategy that favors tasks.

associated with cached graph partitions in each iteration. During execution, the system is more inclined to progress the tasks on cached graph partitions incrementally, rather than completing them all at once. While this method may result in a slight slowdown of individual task speeds, it significantly improves overall execution efficiency and throughput. 3) Fine-Grained Data Sharing Strategy: In traditional strategies, task data access is independent of each other, even if they process identical data. GraphCPP, combining its LLC-level graph partition with asynchronous mechanisms, achieves fine-grained data sharing among graph partitions. When a high-priority graph partition is loaded into the cache, associated tasks are subsequently awakened and enter asynchronous scheduling. This fine-grained data sharing approach markedly reduces redundant data access, thereby enhancing system performance.

In summary, this paper makes the following contributions:

1.It reveals the performance bottleneck caused by redundant data access in existing graph query systems when handling concurrent point-to-point query tasks. It suggests that the similarity in data access among concurrent query tasks can be leveraged to optimize the throughput of concurrent tasks.

2.GraphCPP is implemented, a data-centric concurrent point-to-point query system on dynamic graphs, which optimizes single query speed using the core subgraph mechanism. It then leverages data access similarity between concurrent tasks to accelerate the throughput of concurrent point-to-point query systems.

3.We compare GraphCPP with the current state-of-the-art point-to-point query system XXXXXX. The results demonstrate that XXXXXXXXX

背景和动机

Preliminaries

并发点对点查询、 图分块、 Hub节点 sub-hub节点、 度数 查询节点对 上界值 下界值、 距离值



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。

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。

3，统计大量作业访问数据的重叠性，证明“数据冗余访问”。具体到点对点查询，就是路径在核心子图的重叠

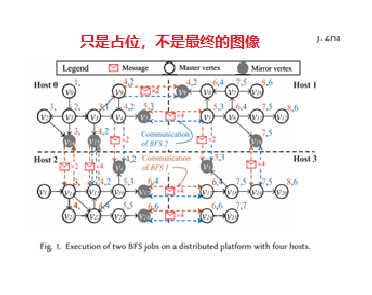
4，统计重叠数据访问占总数据的比例，证明“数据冗余访问”。具体到点对点查询，就是路径在核心子图的重叠

5，统计并行调度缓存错失率，说明并行调度的方案低效的原因。

现有的大多数点对点查询系统是为了优化单次查询速度而设计的，但是如图x所示，我们的统计表明，很多实际应用场景需要应对大规模的并发查询，这类场景对单次查询的速度很宽容，更加重视系统整体的吞吐量。但是如图x所示，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大量的冗余访问，为了定性地证明这一点，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。BACKGROUND AND MOTIVATION

并发任务的冗余数据访问

我用下图的例子，来说明在同一个底层图上执行并发的点对点查询，存在的冗余内存访问。如下图(参照GraphTune)所示，XXXXX结合图像描述。



对图像的修改：抛弃分区，高度顶点和普通顶点要大小和颜色区分。不同查询的路径颜色要区分。图像下面给出每个查询遍历的路径点。

如图x所示，数据重叠访问在并发任务中大量存在，对这部分数据的重复访问属于冗余访问。且如图x所示，每轮查询中冗余的数据访问占到总访问的XXXX。由于少量的高度顶点成为热门的查询路径候选点，它们被不同的查询反复加载。然而，不同任务加载的时间不同，即使在同一时间加载相同数据，在现有系统体系下也不支持这部分数据的共享。如图x所示，这部分数据在LLC中频繁换入换出，导致很高的缓存不命中率，从而导致很差的系统吞吐量。Redundant Data Access Overhead

我们的启发

通过上述的实验，我们观察到了以下几点结论：

**观察1**：图上的高度顶点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。假如可以识别图上的高度顶点Hub，并建立起Hub之间的核心子图，就可以为每次查询快速确定一个近似的距离值。不一定是最优的结果，但是基于这个值我们可以对查询过程执行有效地剪枝，从而大大加快单次查询过程。

**观察2**：不同任务之间的数据访问存在相似性，它们的遍历路径有很大部分是重叠的。这点和观察1是契合的。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细粒度的数据共享，这带来了冗余的数据访问开销。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。

**Our Motivation**

Through the experiments conducted above, we have observed the following conclusions:

Observation 1: Highly connected vertices in the graph are more likely to be traversed repeatedly by different tasks. Different query paths can be regarded as distinct lines, and highly connected vertices serve as intersections of these lines, appearing frequently in various tasks. If we can identify these highly connected vertices, or "Hubs," in the graph and establish a core subgraph among them, we can quickly determine an approximate distance value for each query. While it may not always yield the optimal result, based on this value, we can effectively prune the query process, significantly accelerating individual queries.

Observation 2: There exists similarity in data access between different tasks, as a substantial portion of their traversal paths overlap. This aligns with Observation 1. 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 fine-grained data sharing between tasks, this leads to redundant data access costs. This insight motivates 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 enhance the throughput of concurrent queries.

系统概述

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，我们提出了一个新颖的以数据为中心的高效系统-GraphCPP。它包含一个高效地核心子图查询机制，通过维护常设顶点，来确定路径距离值上界，用于查询过程中的剪枝，从而加快单次查询的速度。更重要的是它包含一个高效地以数据为中心的缓存执行机制，它利用并发任务之间的数据相似性，将多任务共享的图分块加载到LLC缓存，驱动关联任务批量执行，提高了缓存效率，提高了并发系统的吞吐量。

单次查询优化：核心子图查询机制。在查询执行前，我们遍历整个图，统计所有顶点的度数信息。根据度数信息筛选hub节点和sub-hub节点（Dhub> threshold1>Dsub-hub> threshold2），前者记录了所有顶点的索引，后者记录了所有hub（包含sub-hub）节点的索引，两种hub节点共同组成了核心子图。核心子图的作用是通过维护已知顶点之间最短距离值，来为未知的查询顶点对提供一个上界值。这个值不一定是精准的最短距离，但是可以将所有距离值大于它的路径剪枝。通过这样的方式，大大减少了遍历过程的检索空间，提高了单次点对点查询的速度。

并行查询优化：以数据为中心的缓存执行机制。在GraphCPP中图分区从逻辑上进一步划分为LLC大小的图分块。一个关联任务映射机制会统计与每一个分块关联的任务数量。优选关联任务最多的分块，加载到LLC中。一个关联任务触发器会触发与缓存中分块相关联的任务批量执行，这种一次加载，多任务共享的以数据为中心的处理机制显著降低了冗余数据访问，提高了系统的整体吞吐量。

目前，GraphCPP支持PPSP、Viterbi、PPWP、PPNP、BFS、Reachability、Connectivity等一系列点对点查询算法的并发执行。

系统架构

下图展示了GraphCPP的系统框架。GraphCPP以Gemini为benchmark，之所以采用gemini是因为它目前 state-of-art 的分布式内存图计算引擎，具有良好的性能和可编程性。

我们在gemini的基础上添加了细粒度图分块管理模块，关联任务触发模块，细粒度数据同步模块。我们复用了gemini的图分区存储机制，同时引入了一个细粒度的图分块管理模块，它从逻辑上把粗粒度的图分区划分为可以被LLC容纳的细粒度的图分块。该模块采用一个优先级计算公式，利用当前分块的关联任务数量得到当前分块的优先级（关联数量越多，优先级越高）。分块管理模块会调度优先级最高的分块至缓存，期间复用了Gemini的访问。关联任务触发模块，会根据图分块管理模块提供的关联任务信息，触发关联任务批量执行。最后考虑到虽然各个任务访问相同的数据分块，但是不同任务的访问顺序不同，可能导致无法数据共享。数据同步模块采用一种细粒度的同步方式，来实现缓存数据的共享。



整体执行流程

我们将GraphCPP的整体执行流程用伪代码的形式展示。算法接收两个输入参数，分别是当前计算节点所包含的所有图分块的集合*C*，以及当前计算节点所包含的所有查询任务的结集合*Q*。我们首先设置一个动态大小的连续内存空间，用于存放所有的查询任务（第一行）。接下来循环处理，只要当前仍有未结束的查询任务（第二行），GraphCPP就会调用ChoseNextSharingChunk选取当前优先级最高的图分块*ci*。通过统计每个任务的关联分块，我们可以找出所有与当前图分块*ci*有关联的查询任务（第四行）。将*ci*加载至缓存，接下来并行处理所有的关联查询操作*qi*（第五行）。我们调用GraphCPPCompute在当前分区上，对任务*qi*执行并发的点对点查询操作。倘若查询没有结束，我们对查询*qi*的状态进行更新，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*ci*存在关联关系，会将*qi*加入到*Qci*继续查询，否则将新生成的查询保存到任务自身的缓存空间。

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

上述算法展示了GraphCPP中的并行查询优化——以数据为中心的缓存执行机制，其中的GraphCPPCompute函数则使用了单次查询优化——高效地核心子图查询机制。下面的章节将详细介绍两个优化机制。

OVERVIEW OF GraphCPP

xxxxx STRATEGY xxxxx

xxxxx STRATEGY xxxxx

高效地核心子图查询机制

Tripoline最早提出常设顶点(Hub)的概念，通过定期维护“Hub顶点索引”（Hub顶点到其余顶点的距离值以及所有其余顶点到Hub顶点的距离值），来为随时会到来的任意顶点对的查询提供一个近似上界值，从而加快单次索引的计算速度。但是Tripoline的Hub顶点索引机制存在以下缺陷：

缺陷1：Tripoline设计的hub索引中，需要记录hub顶点与其它所有顶点的索引值，而图的规模往往很大，所以建立索引的计算开销和存储开销很大。

缺陷2：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，Hub顶点索引需要基于最新的图快照来进行动态维护。由于hub索引记录hub顶点与每一个顶点的索引关系，这意味着任何更新都会对所有的hub顶点造成影响，所以维护索引的计算开销很大。

需要统计hub顶点的存储开销。

一般来说，为了应对随时到来的随机查询，hub顶点的数量越多，我们越容易找到精确的“上界值”，进而可以加快点对点查询的计算。但是基于上面提到的缺陷，我们不能无限制的增加hub节点的数量，即使我们可以利用闲时算力分摊计算索引的开销，存储和维护的开销仍然不能忽略。而较少的hub节点意味着我们得到的上界值并不准确，对性能的优化效果有限。

基于以上观察，我们提出了基于两级hub索引的核心子图查询机制，如下图所示。具体地，我们在Hub顶点的概念基础上提出了“次级常设顶点（sub-hub）”。传统的Hub顶点索引，需要记录hub顶点与其它所有顶点的索引值。而sub-hub顶点的索引，只需要记录常设顶点（包含Hub顶点和sub-hub顶点）之间的索引值，由于hub顶点的数量远小于总顶点，这部分的存储开销远小于Hub顶点的索引。hub节点和sub-hub节点共同组成了核心子图，利用核心子图进行查询，我们得以使用较少的hub顶点，实现更好地上界值查询。GraphCPP提出了一个高速地核心子图查询机制。它的执行步骤如下：**1，建立常设顶点索引**：它首先遍历所有分区，统计顶点度数。然后将度数大于threshold 1的顶点当做Hub顶点，将度数小于threshold1但是大于threshold2的顶点称为sub-hub顶点（Dhub>threshold1>Dsub-hub> threshold2, threshold1, threshold2的计算参照公式X）。对于所有的hub顶点，我们执行单源最短路径算法计算hub与其他所有顶点的最短距离值。对于所有的sub-hub顶点，我们按照同样的方式计算sub-hub顶点与其他所有常设顶点的最短距离值。建立常设顶点索引是为了应对随时发生的未知查询，相当于利用闲时算力加速查询计算。**2，计算上界**：常设顶点的索引值可以帮助我们迅速找到对应查询的上界值。具体来说，当一个查询到来，遍历所有hub顶点，查询源点到该hub节点的距离值Ps,i，以及从该hub节点出发到达目的顶点的距离值Pi,D，则以该hub顶点为中介的路径的距离值为Pi。Pi不一定是源点和目的顶点之间的最短路径值，但是为我们的遍历提供了参考，我们可以使用其来进行剪枝。**3，剪枝查询**：同SGraph的操作一样，GraphCPP使用了基于上界和下界的剪枝查询策略。把Pi作为剪枝的“上界”，所有距离值大于该上界的路径都被剪枝。同时GraphCPP利用三角不等式，通过常设顶点索引值以及当前的上界值Pi推导出路径的下界。所有考虑下界后路径值大于Pi的路径也会被剪枝。4，**核心子图上的路径加速**：2.2节中统计发现绝大部分点对点查询路径都会包含多个hub节点（既包括hub，又包含sub-hub），这之间也包含了一个潜在的优化机会。我们使用双向遍历算法，如果前向遍历和后向遍历都遍历到了hub节点，就可以通过查询核心子图快速求得两个hub节点之间的最短路径值。虽然这条路径不一定是全局的最短路径，但是可以极大地细化上界值，加速计算速度。



基于两级hub索引的核心子图查询机制：当查询到来时，我们首先根据Hub顶点（h1,h2……）的索引计算出上界值。然后执行双向点对点查询算法，在计算的过程中利用遍历得到的路径值更新上界值，利用上界值对所有无效路径（路径值大于上界）进行剪枝。由于hub节点的数目较少，即使我们刻意选择高度顶点作为hub，仍然会有不少的路径不经过hub，对于这部分路径，需要使用sub-hub节点进一步剪枝。具体地，前（后）想遍历的过程中，会记录路径上遇到的sub-hub。当前后向都至少遇到一个常设顶点时，通过下面的公式得到一个新的上界值。

Ub=min{Ub,(Ps,ki+Pki,kj+Pkj,d)|ki是前向遍历中遇到的sub-hub，kj是后向遍历中遇到的sub-hub}

不断更新的上界值，会形成更加严格的约束条件。此外我们采用了和SGraph一样的下界剪枝策略，配合上界剪枝，减少遍历空间，加速点对点查询的计算过程。

每次查询中前（后）向查询会判断是否查询到常设顶点，当查到一个常设顶点，它会记录当前的前（后）向距离值。当前后向都至少遇到一个常设顶点时，通过查阅sub索引，直接得到两个sub顶点之间的距离值P。此时temp=前向距离值+P+后向距离值。使用temp可以更新上界值，从而加快收敛。

基于以上观察，我们提出了基于两级hub索引的核心子图查询机制，如下图所示。具体地，我们在Hub顶点的概念基础上提出了“次级常设顶点（sub-hub）”。传统的Hub顶点索引，需要记录hub顶点与其它所有顶点的索引值。而sub-hub顶点的索引，只需要记录常设顶点（包含Hub顶点和sub-hub顶点）之间的索引值，由于hub顶点的数量远小于总顶点，这部分的存储开销远小于Hub顶点的索引。hub节点和sub-hub节点共同组成了核心子图，利用核心子图进行查询，我们得以使用较少的hub顶点，实现更好地上界值查询。

以数据为中心的缓存执行模型

在3.2节中我们观察到并发任务之间的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。我们的缓存执行模型，本质上是将原本的线性任务调度顺序，改为细粒度的以数据为中心的调度顺序，从而提高缓存利用效率，提高系统吞吐量。而要实现这样的执行模型，我们需要解决两个问题：1，如何确定共享的数据部分。2，如何实现多任务间的数据共享。下面是我们的实现细节。

如何确定共享的数据部分

1，确定进行共享的图数据粒度。分布式内存系统通过缓存来提升数据访问效率，所以理想情况下共享的图分区需要能完整地载入LLC，从而避免访问分块不同部分带来的频繁换入换出。但是图分区的粒度也不能过于小，否则会增任务处理的同步开销。下面展示了如何确定合适的分块大小。

我们使用CS表示要确定的共享数据分块的大小。使用T表示执行并行计算的线程数。使用GS表示图像的完整结构数据的大小。使用|V|表示图像的顶点总数。使用VS表示存储一个顶点的状态信息平均所需的空间大小。RS是预留的冗余空间的大小。LLCS是LLC缓存空间的大小。下面公式中不等号左边部分被两个“+”划分为了三个部分。第一项是图结构信息的大小，第二项是近似的任务特定信息的大小。由于第二项并非确切值，为了避免数据溢出，第三项预留出了一部分冗余空间。

2，逻辑划分。分布式系统通常采用分区技术来将一个大规模图划分为可以容纳到单台机器的内存中的图分区。GraphCPP在内存容量级别的图分区的基础上进一步地将图划分为细粒度的图分块，和此前划分不同的是，这里的块划分是逻辑划分，而非在物理上划分。清单x展示了GraphCPP划分图分块的伪代码：

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

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 ，一个是记录逻辑划分块的集合chunk\_set。接着定义变量chunk\_edge\_num记录当前分区的边数目。定义变量chunk，它是一个字典，key是源顶点ID，value是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chunk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chunk\_set。这样当分区中的所有数据遍历完一遍，我们就得到了从逻辑上划分的图分块的集合。

3，将查询任务与所属分块关联。上一步中我们采用逻辑划分的方式，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。具体地，每一个查询任务都记录了当前遍历过程中的活跃顶点集，首先通过顶点的ID号反推出其所在的图分块，并将查询到的信息存储在专门的数组中。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。

4，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，优先调度该分块。

通过以上步骤我们产生了一个个供任务共享的图分区，并通过一个经济的优先级调度顺序，将图分区加载到LLC缓存中，接下来还需要细粒度的处理机制来利用这部分共享数据。

如何实现多任务间的数据共享

关联任务触发机制。在上一章节，我们建立起了图分块与其关联任务的映射，并将共享分区加载到了LLC缓存。我们为不同任务分配单独的线程，由于不同线程可能访问相同的数据，可能导致访问冲突。因此将相同的查询的操作进行合并，将合并操作交由同一线程，可以以无原子的方式处理，因为只有一个线程更新查询特定的数据。其次由于线程仅处理相同查询的数据，并且特定于查询的数据存储在连续的内存空间中，因此避免了对其他查询的数据的跨内存访问。

当一轮查询执行完成后，查询任务的状态发生了改变，活跃顶点集也会发生改变，新的活跃顶点可能处于其他的分区，此时需要更新分区与关联查询的映射关系。如果任务仍然有与LLC中关联的查询，可以继续利用当前的分区中的数据，直至与当前分区的所有查询都被执行完毕。

1. 基于优先级的图分区调度策略
   1. 逻辑划分
   2. LLC大小，利用缓存局部性
   3. 生成一个chunk\_table数组来描述每个逻辑chunk的关键信息，用于多个作业共享的图分区的定时访问
2. 基于让步的异步任务执行策略

3，细粒度的数据共享策略

实验评估

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

实验设置

预处理开销

整体性能对比

调度策略性能

是否开启索引子图对结果影响

可扩展性

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，使最短路径计算变得异常困难。它提出了Hub-Network概念，以限制Hub节点的搜索范围。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定位在专用加速器，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通过在日常维护一些“常设顶点”，以常设顶点为“中介”，推导两点之间近似的”上界“，这样实现了”无先验知识“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.

结论

致谢

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.

CONCLUSION

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