目录

[动态图上以数据为中心的并发处理点对点查询系统 2](#_Toc149551447)

[摘要 2](#_Toc149551448)

[前言 3](#_Toc149551449)

[背景和动机 7](#_Toc149551450)

[Preliminaries 7](#_Toc149551451)

[不精确的索引上界 9](#_Toc149551452)

[并发任务的冗余数据访问 9](#_Toc149551453)

[我们的启发 11](#_Toc149551454)

[系统概述 12](#_Toc149551455)

[系统架构 13](#_Toc149551456)

[整体执行流程 14](#_Toc149551457)

[基于核心子图的查询加速机制 15](#_Toc149551458)

[以数据为中心的缓存执行模型 17](#_Toc149551459)

[如何确定共享的数据部分 17](#_Toc149551460)

[如何实现多任务间的数据共享 20](#_Toc149551461)

[实验评估 21](#_Toc149551462)

[相关工作 22](#_Toc149551463)

[结论 23](#_Toc149551464)

[废弃材料 25](#_Toc149551465)

[素材库： 25](#_Toc149551466)

英文翻译

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，加快对单个查询的执行速度；2，优化并发查询任务的执行效率；对于前者，我们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距离值的“全局索引”瘦身为只维护高度顶点之间距离值的“核心子图索引”。由于开销更小，核心子图可以容纳更多高度顶点，增大了覆盖查询最短路径的概率，意味着可以提高更精确的上界值，从而可以提升剪枝的效果，加速查询过程；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 Shortest 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性能，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；对于前者，目前主流的解决方案是通过剪枝来减少查询过程对顶点的遍历，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；我们发现剪枝的关键是尽快确定更加精确的上界。之前的方案中，为了减少处理维护的开销，普遍选用少量高度顶点来维护全局索引。由于图符合二八定律，从少量高度顶点出发的索引表可以为大部分查询提供上界值，但是代价是无法确保精确性。对于后者，我们发现当前的点对点查询并没有考虑高并发的点对点查询场景，而我们在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。总之，由于上述原因，现有的点对点查询系统存在冗余的数据访问开销，常常造成严重的性能瓶颈。

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.

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。

**单次点对点查询优化**：GraphCPP提出了一个基于核心子图的查询加速机制，它的核心思想是：使用轻量的索引扩大索引的覆盖范围，从而为查询提供更加精确的上界。在传统的全局索引机制中，由于需要负责维护从高度顶点到图中所有顶点的距离值，它的存储开销，计算开销以及维护开销都很大，所以现有的方案不约而同地选择限制索引顶点的数目。如Tripoline和SGraph都选择只采用16个高度顶点建立索引，这限制了索引顶点的效果，导致无法提供精确的上界值。而核心子图采用了更轻量级的索引，即索引只需要维护图中高度顶点之间的距离值，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短距离。这样一来大大减少了索引本身的开销，从而可以选择更多的高度顶点加入到核心子图中。更多的索引会覆盖更多的路径选择，增大了覆盖两点之间最短的那一条路径的概率，减少了冗余顶点的访问，从而加速单次查询速度。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图结构数据访问的共享，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将查询过程中的数据解耦成 “任务特定数据”和“图结构数据”。前者与特定查询任务相关，需要由任务独自保存。后者只与数据本身相关，可以被多个任务共享。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. 揭示了现有点对点查询系统全局索引机制的高开销，并对此提出了一种基于核心子图的轻量级索引机制。指出当前工作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. 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，它利用基于核心子图的查询加速机制加快单次查询速度。然后利用并发任务之间的数据访问相似性，优化并发点对点查询系统的吞吐量。
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

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

现有的点对点查询方案一个关键的思想是“建立全局索引，并通过剪枝来减少对冗余顶点的访问”。它们从少量高度顶点出发，建立起到达全图的索引。进而利用图上的三角不等式来获得距离的近似值，也就是上界。利用上界值，可以在查询过程中对冗余顶点访问进行剪枝。这种机制相当于利用了闲时算力，来加快瞬时查询需求的响应速度。然而我们发现即使在最先进的点对点查询系统中，预估的上界值和最终的计算结果也相差很大。这意味着使用索引获得的上界是很不精确的，而使用不精确的上界值会削弱剪枝的效果，导致冗余访问。此外，我们发现现有的大多数点对点查询系统是为了优化单次查询而设计的，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。然而如图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。

本章分为四个部分，我们首先介绍了并发点对点查询中的一些概念；其次我们分析了现有索引剪枝机制的弊端；第三，我们分析了当前并发点对点查询方案的痛点，最后我们描述了从观察中获得的启发。

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，例如对于最短路径查询Rib表示Si和Di之间的最短路径。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。我们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。

定义五：上界和下界：在点对点查询中，我们采用了基于“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。其中上界UB表示当前已知的从源点到目的顶点的最短路径的距离值，所以在遍历一条路径时，如果发现当前的路径距离大于UB则需要被剪枝，如果新发现一条路径距离小于UB，则更新UB的值。下界LB表示从当前顶点v到目的顶点保守的最短距离预测值，即预测的LB小于或等于顶点v到目的顶点实际的最短距离，它是通过对索引运用三角不等式推导出的。如果一条路径加上LB的值比UB大，则这条路径一定比已有的路径差，需要被剪枝。

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距离值，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距离值。

不精确的索引上界

所需图像：  
1，统计在不同数据集中，索引的精确度变化。

2，索引精确度对于活跃顶点访问数目的影响

当前的点对点查询都沿袭了Tripoline中的全局索引机制，即从全部顶点中选取度数最高的K个顶点统计到达全图的最短距离值。对于K的选择，Tripoline和SGraph都采用了一个固定数目（默认是16）。由于不同的图规模不同，分布特征不同，这样的选择显然是缺少灵活性的。如图xxx，在不同规模的数据集中，即使选取同样数目的常设顶点，索引获取的上界的精确度差异也很大。 而图xxx表明，而失准的索引值会弱化剪枝的效果，造成冗余的数据访问。精准的上界值则可以大大减少激活顶点数，加速计算过程。

并发任务的冗余数据访问

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。

1. 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

**Inexact Index Upper Bound**

**Redundant Data Access Overhead**

下图展示了在同一底层图上执行不同的并发查询的例子。为了简便起见，我们没有画出顶点之间全部的连接边，也没有展示查询过程对冗余路径的访问。图中不同颜色的带箭头路线表示不同查询任务遍历的路径，灰色圆点则表示图上的普通顶点，红色圆点表示图上的高度顶点。可以看出不同查询遍历的路径存在重叠部分，包含索引顶点的路径重叠概率更高。我们对并发查询中的冗余访问做了定量分析，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。

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**：现有的全局索引机制无法提供精确的上界值，从而弱化了剪枝效果，增加了对冗余数据的访问开销。

**观察2**：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，并利用闲时算力，在查询未到来时建立起hub之间的索引，就可以为每次查询快速确定一个近似的距离值。它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3**：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2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并发点对点查询系统-GraphCPP。它包含一个基于核心子图的查询加速机制，通过维护轻量级的索引顶点，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。此外，它还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，提高了缓存效率和并发点对点查询的吞吐量。

单次查询优化——基于核心子图的查询加速机制。具体地，在计算开始前，GraphCPP统计所有顶点的度数并排序，选择排名[1,k]作为全局索引顶点，它们维护了与图中所有顶点的距离索引，接着选择排名（k,m]的顶点作为核心子图上的顶点（0<k<m<|V|，k和m由用户指定），它们维护了核心子图上顶点之间的距离索引。当查询到来时，首先采用传统的全局索引机制获得粗略的上界和下界，并依次开启剪枝查询。在查询的过程中，系统会判断当前的顶点是否属于核心子图。如果是，则表示查询接入了高速地核心子图，通过查询核心子图的索引，系统会获得从当前高度顶点出发到达其他所有高度顶点的距离值，并将这些高度顶点加入下一轮的查询边界。实际两个高度顶点之间的路径可能经过许多跳，借助核心子图，它们被缩短为了一步。这样加速了查询的进度，可以更快获得精确地上界值，从而减少冗余顶点的访问，从而加速单次查询速度。

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，一个细粒度的任务同步机制负责实现多任务共同访问的图结构数据的共享，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。

GraphCPP Overview

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的访问接口。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。

**System Architecture**

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（所谓分块关联任务，指任务在当前分块存在活跃顶点），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*，回到第5行继续查询，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 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。

xxxxx STRATEGY xxxxx

xxxxx STRATEGY xxxxx

基于核心子图的查询加速机制

Tripoline最早提出索引顶点(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，而当图的规模非常大时，建立索引的计算开销和存储开销也会很大。

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更新。由于hub索引记录hub顶点与每一个顶点的索引关系，这意味着流图的任何更新都会对所有的hub顶点索引造成影响，所以维护索引的计算开销很大。

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顶点的数量，即使我们可以利用闲时算力分摊一部分计算索引、维护索引的开销。而当常设顶点的数量过少，由于图上不同部分的连通性问题，少量常设顶点可能无法覆盖到大部分顶点，意味着无法提供有效地索引值，导致剪枝效果差，增加了遍历开销。

对此，本文在全局索引的基础上，提出了轻量级的核心子图索引。和全局索引相比，它的筛选阈值更小，数量更多，因此可以做到更高的覆盖率，提供更精确的上界值。同时它不再维护全局索引，而只需要维护高度顶点之间的索引，因此它的开销远小于全局索引。清单XXXX展示了核心子图查询的伪代码

基于核心子图的查询加速机制的执行步骤如下：1，建立全局索引（第1-4行）。我们采用了跟SGraph相同的策略计算全局索引。系统在对顶点的度数进行排序之后，选择度数最高的k个顶点（k值由用户确定）。执行SSSP算法计算k个高度顶点与图上的所有顶点的最短路径（包含距离值和路径父节点），将结果存入以高度顶点id为索引的数组保存。2，建立核心子图索引（第5-8行）：放宽筛选的度数标准，选择更多的高度顶点加入核心子图中。由于全局索引顶点已经记录了到达全局的索引，所以要剔除掉这部分顶点。此外，建立好全局索引后，我们可以直接用基于上界和下界剪枝的点对点查询求得核心子图上各点之间的最短路径。3，查询加速（第9-15行）：执行点对点查询，首先借助全局索引确定粗略的上下界值。随后开始剪枝查询。正常情况下，系统会遍历当前顶点的每一个出边顶点，依次对每个顶点的距离值进行剪枝判断，以确定下一轮的活跃顶点。若当前查询的顶点属于核心子图则除了访问出边邻居，还要访问与该顶点相连的其它所有的高度顶点。正常情况下，这些高度顶点之间的状态传播可能需要很多跳才能完成，有了核心子图，可以一步完成这些点之间的状态传播。除了可以更快地完成状态传播，一个隐含的因素是，核心子图上遍布高度顶点，它们更可能出现在两点之间的最短路径上，核心子图可以加快路径的发现过程。4，查询终止（第16-20行）：运用上下界查询技术进行剪枝查询，对于单向查询，从源顶点出发，当遍历到目的顶点时表示发现了一条路径。对于双向查询，两个方向的查询相遇表示发现了一条路径。当新的路径值小于当前上界，则将其更新为新的上界。若路径值大于当前上界，则会被剪枝。发现一条路径并不意味着迭代的结束，我们还需要判断图中的活跃顶点，只有当所有可能的路径都被尝试过，此时的上界被更新为最短的路径值，所有的点的出边路径值都比现有的上界大，活跃顶点的数目降至0，则此时迭代结束。通过上述步骤，我们用轻量级的核心子图索引，更快更精确地获得了查询上界，从而加速单次查询效率。

|  |
| --- |
| Algorithm: Core-Subgraph Query Acceleration. |
| 1: func. BuildGlobalIndex(G, k)  2: vertices = SortAndSelectTopK(G)  3: global\_index = ComputeSSSP(G, vertices)  4: return global\_index  5: func. BuildCoreSubgraphIndex(G, global\_index)  6: core\_vertices = SelectRelaxedMinusGlobal(G, global\_index)  7: ComputePaths(core\_vertices)  8: return core\_subgraph\_index  9: func. QueryAcceleration(G, source, target)  10: bounds = GetBoundsUsingGlobalIndex(source, target)  11: active\_vertices = [source]  12: while has\_active(active\_vertices) do:  13: active\_vertices = UpdateActiveVertices(G, bounds, active\_vertices)  14: end while  15: return shortest\_path  16: func. QueryTermination(G, upper\_bound)  17: while not PathFound(G) do:  18: path = FindAndUpdatePath(G, upper\_bound)  19: end while  20: return upper\_bound |



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，否则会增加任务处理的同步开销。下面展示了综合考虑分块图结构数据和任务特定数据，如何确定合适的分块大小。

我们使用CS表示要确定的共享的细粒度数据分块的大小，使用GS表示每个图分区上的图结构数据的大小，则𝛼表示共享图分块部分占分区图像的比例。我们使用|V|表示分区上图的顶点总数，则𝛼⨯|V|表示共享分块所拥有的顶点数目的近似值。我们使用VS表示存储一个顶点的状态信息平均所需的空间大小，则代表了查询任务在共享分块上存储任务特定数据所需空间的最大值。考虑到多核处理器多个核心并发执行，所以缓存中需要保留多个查询的任务特定信息，我们使用N表示执行并行计算的线程数，则表示在缓存中存放当前分块的关联任务的任务特定数据所需要的空间。RS是预留的冗余空间的大小。LLCS是LLC缓存空间的大小。则在满足下列不等式的前提下，的最大值就是图分块的大小。

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，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。

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，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。

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

触发关联任务并发执行。每个查询任务*qi*在执行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。在上一章节介绍了逻辑上划分图分块，每个分块对应一个chunk字典，它记录了本分块中顶点的id以及本分区中顶点的度数。如果任务*qi*的活跃顶点出现在某个分区的字典中，代表该任务是对应分区的关联任务。利用chunk字典和活跃顶点集Setact,i，我们可以快速确定载入LLC中的活跃分块的关联任务并发执行。如算法X所示，关联任务执行一轮后，各自产生新的活跃顶点。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。

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

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通过在日常维护一些“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.

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

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。