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)

动态图上高效缓存的并发处理点对点查询策略

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，大量点对点查询作业在同一个底层图上并发运行。然而现有的图上点对点查询系统致力于加快对单个查询的响应速度，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高效缓存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，不同查询所需访问的图结构数据分块可能重叠。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，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。

当前问题：摘要只写了数据共享这一个创新点，显得单薄。

GraphCPP: A Cache-Efficient Approach for Concurrent Point-to-Point Queries in Dynamic Graphs

Abstract

As graph processing techniques become increasingly prevalent in domains such as map navigation and network analysis, the demand for concurrent point-to-point query tasks on dynamic graphs has surged. While many contemporary systems prioritize optimizing the response time of individual queries, their efficiency diminishes in high-concurrency query scenarios due to recurrent redundant data accesses. Identifying a consistent pattern in data access across concurrent query tasks, we propose a novel caching mechanism designed specifically for concurrent point-to-point queries. We classify data accessed during graph querying into two distinct types: "graph structural data," which encapsulates the graph's topology, and "task-specific data," which monitors the portions of the graph structural data accessed during queries. Noting that data chunks from separate queries may intersect, our methodology adopts a data-driven scheduling approach. In this context, a single instance of the graph structural data is preserved in memory/LLC throughout the execution of concurrent tasks. These tasks access data cohesively at a detailed level, founded on graph data segments, ensuring a single data access serves multiple tasks. This strategy reduces data access overhead and boosts the throughput of concurrent graph queries. To ascertain the effectiveness of GraphCPP, we conducted comparative evaluations against leading point-to-point query systems, specifically SGraph[x], Tripoline[x], and Pnp[x]. Our experimental data reveals that GraphCPP enhances the efficiency of concurrent point-to-point queries by a significant 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（举例子XXXXXXXXX）；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（举例子XXXXXXXXX）；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（举例子XXXXXXXXX）；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。

然而现有的解决方案无法很好地应对并发需求带来的高负载。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；我们调研了现有的点对点查询方案：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。 这些工作都聚焦于加速单次查询的效率，而忽略了并发任务的优化潜力。

为了解决以上难题，本文提出了GraphCPP，一种新奇的“高效缓存的并发处理点对点查询策略”。它

包含以下几部分：1，基于优先级的图分区调度策略：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，并进一步提出了一种优先级计算方法。

INTRODUCTION

此方法旨在紧密地将查询任务与其相关的图分区联系起来。分区所关联的任务数量增加会导致其优先级提升，使其更可能

被优先缓存，从而加速计算并提升整体效率；2，基于让步的异步任务执行策略：传统的策略倾向于确保单一任务的快速完成，这种方法可能导致部分任务过早完成，而其他任务长时间等待，制约了大规模并发查询的吞吐量。GraphCPP提出了一种基于让步的异步策略，该策略在每次迭代中优选与已缓存图分区关联的任务。在执行过程中，系统更倾向于在缓存的图分区上推进任务的进度，而非一次性完成。这种方法虽可能导致个别任务速度稍减，但显著提高了整体执行效率与吞吐量；3，细粒度的数据共享策略：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结合其LLC级图分区与异步机制，实现了细粒度的图分区数据共享。当具有高优先级的图分区被加载到缓存，关联任务随即被唤醒并进入异步调度。这种细粒度的共享方式显著降低了数据访问的冗余，从而提升了系统的性能。

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。并揭示了并发查询任务之间的数据访问相似性。
2. 实现了GraphCPP，一个动态图上高效缓存的并发处理点对点查询系统，它利用并发任务之间的数据访问相似性，加速点对点查询系统的吞吐量。同时还因为了一些优化措施，优化查询时的剪枝步骤，减少不必要的计算量。、
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

背景和动机

通过数据说明问题急迫性

说明我们从问题中看到的启发

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。我们发现并行任务成为了现有点对点图查询系统的瓶颈。

点对点查询

现有系统处理点对点查询的弊端

我们的系统

需要图像：

1. 统计并行查询执行时间，说明并行执行效率很差。
2. 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。
3. 统计大量作业访问数据的重叠性，证明“数据访问相似性”。

我们观察到大量并发的点对点查询任务之间存在一种“数据访问相似性”：它们执行过程中需要访问的数据有很大的重叠。但是在现有的多任务调度策略下，每个任务在不同时刻重复调度数据，即使这些数据是完全一样的，这带来了严重的冗余数据访问。

BACKGROUND AND MOTIVATION

系统概述

基于xxxxx的优化策略

实验评估

实验设置

预处理开销

整体性能对比

调度策略性能

可扩展性

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。

专用加速器，它使用高度顶点作为中心来加速 PPSP 查询。

，没有并行

并发图计算系统，大部分是核外。内存的forkgraph是单机的。缺少分布式内存优化

结论

致谢

**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。但是该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。

和之前的工作相比：

OVERVIEW OF GraphCPP

xxxxx STRATEGY xxxxx

EXPERIMENTAL EVALUATION

RELATED WORK

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.