目录

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。然而我们的实验表明当前的图查询系统聚焦于优化单次点对点查询的速度，而忽视了对并发点对点查询整体性能的优化。我们观察到由于图数据存在幂律分布的特点，不同的查询的遍历路径往往在少量高度顶点组成的局部路径上彼此重叠，体现出了并发点对点查询任务的数据局部性，这启发我们提出一个数据驱动的并发点对点查询系统-GraphCPP。它将图结构数据划分为LLC 级别的分块，按照每个分块关联的活跃顶点数制定优先级，来依次载入活跃分块，触发关联查询任务的并发执行，实现了**数据共享**，提高了数据访问效率；同时它通过核心子图机制对图中高度顶点之间的距离值进行预计算，以便在查询到来时快速确定高频共享路径段的距离值，实现了**计算共享**，加快查询结果的收敛。此外，我们通过预测任务的查询路径，在调度时从任务池中选择相似任务批量执行，更好地利用了并发点对点查询任务的数据局部性。我们将 GraphCPP 与最先进的点对点查询系统进行对比，包括 SGraph[x]、Tripoline[x]、Pnp[x]，实验表明，GraphCPP 将并发点对点查询的效率提升了 xxxx 倍。

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

Abstract

With the widespread application of graph processing techniques in fields such as map navigation and network analysis, a large number of concurrent point-to-point query tasks running on the same underlying graph pose stringent demands on the throughput of graph query systems. However, our experiments indicate that current graph query systems focus on optimizing the speed of individual point-to-point queries, while overlooking the overall performance optimization for concurrent point-to-point queries. We observe that due to the power-law distribution characteristic of graph data, the traversal paths of different queries often overlap on local paths composed of a small number of high-degree vertices, demonstrating the data locality of concurrent point-to-point query tasks. This inspires us to propose a data-driven concurrent point-to-point query system - GraphCPP. It partitions the graph structure data into LLC-level blocks and prioritizes loading active blocks based on the number of associated active vertices, thereby triggering concurrent execution of associated query tasks, facilitating data sharing, and improving data access efficiency. Simultaneously, it employs a core subgraph mechanism to pre-calculate distance values between high-degree vertices in the graph, enabling rapid determination of distance values for frequently shared path segments upon query arrival, achieving computational sharing and expediting the convergence of query results. Additionally, by predicting the query paths of tasks, we select similar tasks in batches from the task pool during scheduling, better utilizing the data locality of concurrent point-to-point query tasks. We compare GraphCPP with state-of-the-art point-to-point query systems, including SGraph[x], Tripoline[x], and Pnp[x]. Experimental results demonstrate that GraphCPP improves the efficiency of concurrent point-to-point queries by a factor of xxxx

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

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

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

INTRODUCTION

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

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

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

针对上述问题，我们设计了 GraphCPP：一种数据驱动的并发点对点查询系统。针对并发任务数据共享难题，它提出了一种数据驱动的缓存执行机制，将传统的“任务→数据”的调度方式改为“数据→任务”的调度方式，进而实现多任务之间重叠图结构数据访问的共享。在这种执行机制下，GraphCPP 会首先确定要数据的调度顺序，它将图结构数据从逻辑上划分为 LLC 级别的细粒度分块。接着根据查询任务活跃顶点集所处的图分块，将查询任务与其相关的图分块联系起来，任务的活跃顶点每轮都会变化，共享分块的关联任务数也需要每轮更新。统计分块的关联任务信息，关联任务数量越多的分块优先级越高，更可能被优先调度；为了实现“数据→任务”的调度方式，GraphCPP 采用了一种关联任务触发机制。它按照优先级顺序将图分块加载到 LLC 中，并利用每一轮统计得到的任务与数据分块的关联信息，触发当前当前分块的关联任务批量执行，实现了对共享数据的高效访问；针对计算共享难题，GraphCPP 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”。两个高度顶点之间的最短路径可能有很多跳，而核心子图相当于给所有互通的高度顶点增加了一条跳边，边的长度就是两点之间的最短距离。这样，当查询到一个高度顶点时，程序就可以像访问邻居节点一下，访问所有其它的高度顶点，从而实现重叠路径的计算共享。瘦身后的核心子图索引开销远小于全局索引，从而可以选择更多的高度顶点加入到核心子图中，增大了高频共享路径的覆盖范围，提高了计算共享的性能。此外，我们还通过预测不同查询任务的遍历路径，优先调度高度重叠的查询任务批量执行，进一步提高了并发查询的性能。

Addressing the aforementioned challenges, we introduce GraphCPP, a data-driven concurrent point-to-point query system tailored for dynamic graphs. To tackle the issue of data sharing among concurrent tasks, it presents a data-driven caching execution mechanism, transforming the conventional "task→data" scheduling approach into a "data→task" scheduling paradigm, thereby enabling overlapping graph structure data access across multiple tasks. In this execution paradigm, GraphCPP initially determines the scheduling order for data, logically partitioning graph structure data into fine-grained blocks at the LLC level. Subsequently, based on the graph block in which the active vertex set of the query task resides, it associates the query task with the relevant graph block. As the active vertices of tasks change in each round, the number of associated tasks for shared blocks needs to be updated accordingly. By recording the associated task information of blocks, blocks with a higher number of associated tasks are given higher priority and are more likely to be scheduled first. To implement the "data→task" scheduling approach, GraphCPP employs an associated task triggering mechanism. It loads graph blocks into the LLC in priority order and utilizes the task-data association information obtained in each round to trigger the batch execution of tasks associated with the current block, achieving efficient access to shared data. To address the challenge of computational sharing, GraphCPP introduces a query acceleration mechanism based on core subgraphs. It trims the conventional "global index" that maintains distance values for all vertices to a "core subgraph index" that only maintains distance values between high-degree vertices. The shortest path between two high-degree vertices may involve multiple hops, while the core subgraph effectively adds a direct edge between all interconnected high-degree vertices, with the edge length representing the shortest distance between the two points. As a result, when querying a high-degree vertex, the program can access all other high-degree vertices similar to visiting neighboring nodes, enabling computational sharing across overlapping paths. The streamlined core subgraph index incurs significantly lower overhead than the global index, allowing for the inclusion of more high-degree vertices in the core subgraph, expanding the coverage of frequently shared paths, and enhancing computational sharing performance. Additionally, by predicting the traversal paths of different query tasks, we prioritize the scheduling of tasks with high overlap for batch execution, further boosting the performance of concurrent queries.

本文主要做出了如下贡献：

1，分析了现有点对点查询系统处理并发点对点查询任务时冗余数据访问带来的性能瓶颈，并提出利用并发查询任务之间的数据访问相似性优化并发任务吞吐量。

2，开发了 GraphCPP，一个动态图上数据驱动的并发处理点对点查询系统，实现了并发任务之间的数据共享和计算共享，并提出了一个相似任务批量执行策略。

3，我们将 GraphCPP 与当前最先进的点对点查询系统 XXXXXX 进行对比，结果表明 XXXXXXXXX

This paper makes the following contributions:

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

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

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

背景和动机

所需图像（还没画，占位）

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

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，统计重叠数据访问占总数据的比例，证明“数据冗余访问”。

4，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。

5，统计高度顶点占冗余数据的比例，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。

现有的解决方案聚焦于加速单次查询的速度，如 PnP 使用基于下界的剪枝方法来减少查询过程中的冗余访问；Tripoline 通过维护中心顶点到其它顶点的日常索引，实现无需先验知识的快速查询；SGraph 利用三角不等式原理，提出了基于“上界+下界”的剪枝方法，进一步减少点对点查询过程中的冗余访问；然而如图 x 所示，我们的统计表明，图上的并发点对点查询正在成为原来越迫切的需求，它们更重视并发查询任务的吞吐量，对于单次查询的速度则比较宽容。如图 x 所示，我们证明现有系统在处理大规模并发查询时吞吐量很差。这种坏结果出现的原因是并发任务之间存在对图结构数据大量的冗余访问。为了定性地分析上述问题，我们在 XXXXX（机器配置），选取了 XXXXX（现有最佳方案），在 XXXXX（图数据集上），进行并行点对点查询的性能评测。

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

BACKGROUND AND MOTIVATION

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

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

Preliminaries

定义一：图：我们使用 $G=(V,E)$来表示有向图，其中 V 是顶点的集合，E 是由 V 中顶点组成的有向边的集合（无向图中的边可以被拆分为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 $P_i=(V_{Pi},E_{Pi})$来表示有向图的第 i 个图分区，使用 V_{Pi} 表示图分区中顶点的集合，E_{Pi} 是由 V_{Pi} 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 P_i 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 $q_i=(S_i，D_i)$表示任务 i 对应的查询。其中 S_i 和 D_i 分别表示查询 q_i 对应的源顶点和目的顶点。查询 q_i 得到的结果值为 R_{SD}，对于不同的算法，它有着不同含义，例如对于最短路径查询 R_{ib} 表示 S_i 和 D_i 之间的最短路径。我们使用 $Q=\{q_1,q_2,\ldots q_{|Q|}\}$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 h_i（$i\in[1,k]$，k 值由用户指定，一般设为 16），$d_{i,j}$（$V_j\in V$）表示从索引顶点 h_i 出发到达图中任意顶点 V_j 的距离，当两点之间不存在可达路径，该值设为极大值。同理 $d_{j,i}$（$V_j\in V$）表示从图中任意顶点 V_j 出发到达索引顶点 h_i 的距离。无向图的 $d_{i,j}$ 和 $d_{j,i}$ 是相等的。

定义五：上界和下界：在点对点查询中，上界 UB 表示当前已知的从源点到目的顶点的最短路径的距离值，下界则 LB 表示从当前顶点 v 到目的顶点保守的最短距离预测值，预测的 LB 小于或等于顶点 v 到目的顶点实际的最短距离。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，它们本质上也是一种计算共享。

Preliminaries

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

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

Definition 3: Point-to-Point Query. We use $q_i=(S_i,D_i)$ to represent the query corresponding to task i. Here, S_i and D_i respectively denote the source and destination vertices of query q_i. The result value obtained by query q_i is represented as R_{SD}. For different algorithms, it holds different meanings. For example, for shortest path queries, R_{ib} represents the shortest path between S_i and D_i. We use $Q=\{q_1,q_2,\ldots,q_{|Q|}\}$ to represent the set of concurrent point-to-point queries, where $|Q|$ denotes the total number of queries.

Definition 4: Index. The index records the distance from a specific vertex to other vertices. We select the top k vertices with the highest degrees in the graph as index vertices $h_i(i\in\left[1,k\right]where\ k\ is\ user-specified,\ typically\ set\ to\ 16)$. $d_{i,j}(V_j\in V)$ represents the distance from index vertex h_i to any vertex V_j in the graph. When there is no reachable path between two vertices, this value is set to a large value. Similarly, $d_{j,i}(V_j\in V)$ represents the distance from any vertex V_j to index vertex h_i. For undirected graphs, $d_{i,j}$ and $d_{j,i}$ are equal.

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

定义六：核心子图：和索引类似，核心子图也会筛选出图上的高度顶点，但是它筛选的阈值更低，意味着有更多顶点可以被选中。这些高度顶点彼此相连组成核心子图，图上两个高度顶点之间边的权重代表两个点之间的距离值，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。

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

并发点对点查询任务的性能瓶颈

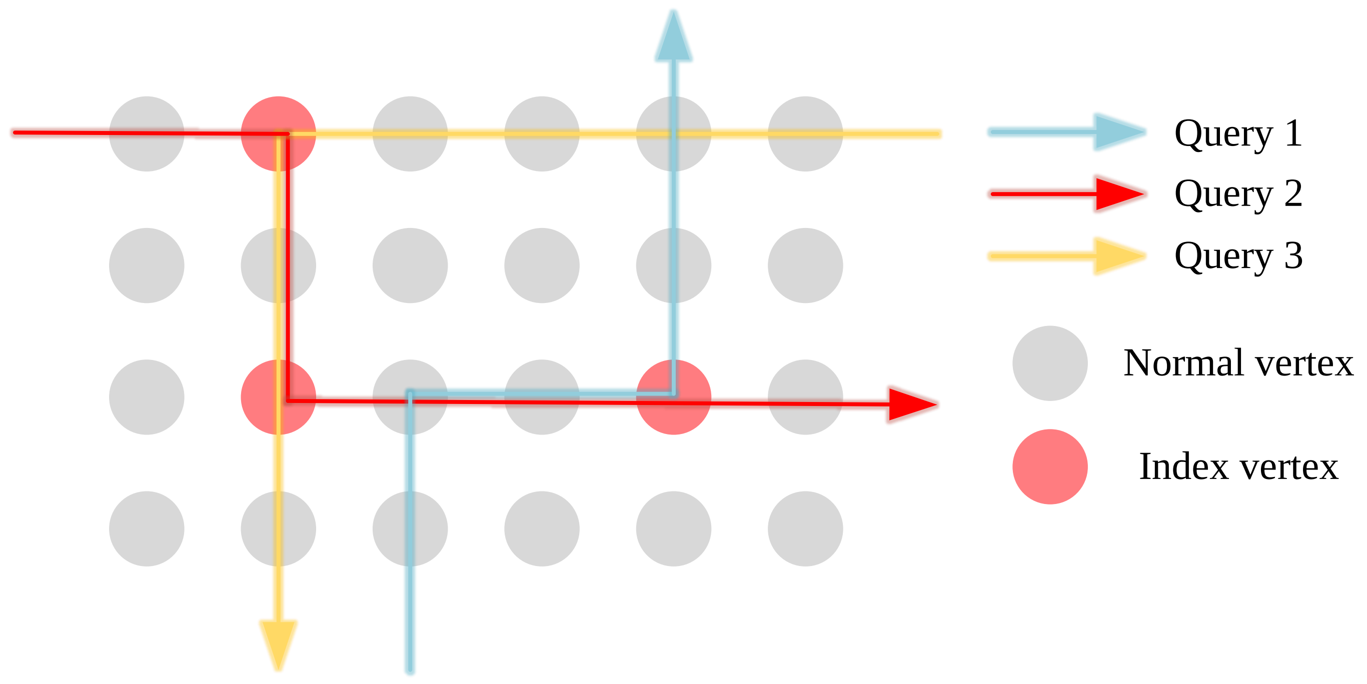
Performance Bottlenecks in Concurrent Point-to-Point Query Tasks

并发任务的冗余数据访问开销

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。

Redundant Data Access Overhead in Concurrent Tasks

The diagram below illustrates examples of performing different concurrent queries on the same underlying graph. For simplicity, we did not depict all the connecting edges between vertices, nor did we show the access to redundant paths during the querying process. Different colored arrowed paths in the figure represent the paths traversed by different query tasks. Gray circles represent regular vertices in the graph, while red circles denote high-degree vertices. It can be observed that there are overlapping portions in the paths traversed by different queries, with a higher probability of overlap when it involves index vertices. We conducted a quantitative analysis on the redundant access in concurrent queries. As shown in Figure x, data overlap access is prevalent in concurrent tasks, constituting redundant access to this portion of data.



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。

并发任务的冗余计算开销

如图 X，我们发现任务间重叠路径访问有很大一部分属于高度顶点，这意味着不同任务会重复计算高度顶点之间的距离值，这带来了极大的冗余计算开销。一些工作尝试使用全局索引，维护少量高度顶点到图中其它顶点之间的距离值，从而避免重复计算到这些高度顶点之间的距离值。但是这种处理方式显然会带来极大的开销，以 Tripoline 为例：全局索引会以每个高度顶点为起点执行 SSSP 算法，并存储高度顶点到达所有顶点的距离值，每个距离值占 4 个字节（有向图需要记录正反两个方向的距离值则需要 8 个字节）。在动态图场景，当发生了图突变，需要更新每一个收到影响的索引顶点。计算、存储、维护开销都会随着图规模的增大和高度顶点数目的增加而成倍放大，所以实际使用时，全局索引的值通常很小（在 Tripoline 和 SGraph 中这个值都设为 16）。如图 X，我们证明固定数目的全局索引顶点所包含的多任务重叠路径的比率很低，这弱化了计算共享的效果。

Redundant data access in each round of queries accounts for approximately XXXX% of the total access. Due to a small number of high-degree vertices becoming popular candidates for query paths, they are repeatedly loaded by different queries. However, the loading times vary among different tasks, and even if the same data is loaded at the same time, the existing system architecture does not support the sharing of this portion of data. As shown in Figure x, this data is frequently swapped in and out of the LLC, resulting in a high cache miss rate and consequently, a poor system throughput.

Redundant Computational Costs

As shown in Figure X, we observe that a significant portion of the overlapping path accesses between tasks involve high-degree vertices. This implies that different tasks redundantly calculate the distance values between high-degree vertices, resulting in substantial redundant computational costs. Some approaches attempt to use a global index to maintain the distance values from a small set of high-degree vertices to all other vertices in the graph, thus avoiding redundant calculations to these high-degree vertices. However, this approach evidently introduces significant overhead. Taking Tripoline as an example: the global index performs the SSSP algorithm starting from each high-degree vertex, and stores the distance values from the high-degree vertices to all vertices. Each distance value occupies 4 bytes (in the case of directed graphs, recording distance values in both directions requires 8 bytes). In dynamic graph scenarios, when a graph mutation occurs, every affected index vertex needs to be updated. The computational, storage, and maintenance costs increase exponentially with the growth of the graph scale and the number of high-degree vertices. Therefore, in practical use, the value of the global index is typically set to a small number (both Tripoline and SGraph set this value to 16). As shown in Figure X, we demonstrate that a fixed number of global index vertices have a low coverage over overlapping paths, which weakens the effect of computational sharing.

我们的启发

通过上述的观察，我们得到到了以下启发：

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

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

Our Motivation

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

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

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

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，我们提出了一个数据驱动的高效并发点对点查询系统-GraphCPP。它通过一个高效地以数据为中心的缓存执行机制，利用并发任务之间的数据相似性，实现了一次数据加载，多个任务共享。它还通过一个基于核心子图的查询加速机制，建立了高度顶点之间的距离索引，实现不同任务之间高频重叠路径的计算共享。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。

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。

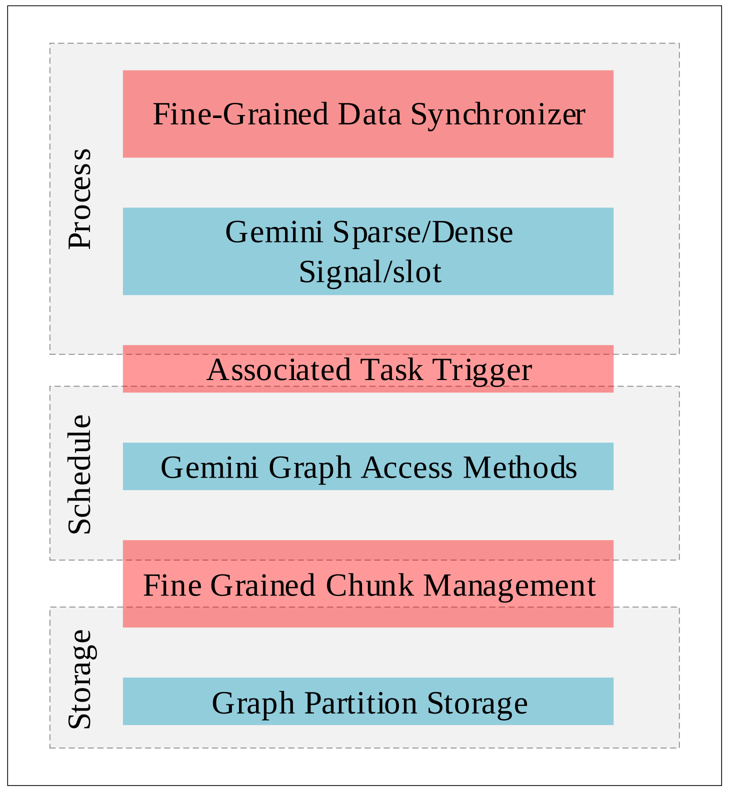
GraphCPP Overview

To enhance the execution efficiency of concurrent point-to-point queries on dynamic graphs, following a detailed examination of the computational intricacies, we propose a data-driven efficient concurrent point-to-point query system, GraphCPP. It employs a cache-centric execution mechanism that is centered around data, enabling multiple tasks to share the results of a single data load by capitalizing on the data similarity between concurrent tasks. Additionally, it establishes a distance index between high-degree vertices through a core-subgraph-based query acceleration mechanism, facilitating shared computation of high-frequency overlapping paths among different tasks. Furthermore, it leverages path prediction to drive the batch execution of similar queries with overlapping paths, further exploiting data similarity.

System Architecture

The figure below illustrates the system architecture of GraphCPP. We chose Gemini as the benchmark due to its current status as a state-of-the-art distributed memory graph computing engine, known for its commendable performance and programmability.

Building upon Gemini, we incorporated a fine-grained graph partition management module, an associated task triggering module, and a fine-grained data synchronization module. While reusing Gemini's graph partition storage mechanism, we introduced a fine-grained graph partition management module. This module logically subdivides the coarse-grained graph partitions into finer-grained units that can be accommodated by the Last Level Cache (LLC). It employs a priority calculation formula to determine the priority of the current partition based on the number of associated tasks (higher association count leads to higher priority). The partition management module schedules the partition with the highest priority to the cache, leveraging Gemini's access interface in the process. The associated task triggering module, relying on the task information provided by the partition management module, triggers the batch execution of associated tasks. Finally, considering that different tasks may access the same data block in different sequences, which could hinder data sharing, the data synchronization module employs a fine-grained synchronization approach to facilitate shared access to cached data.


Process
Fine-Grained Data Synchronizer
Gemini Sparse/Dense Signal/slot
Associated Task Trigger
Schedule
Gemini Graph Access Methods
Fine Grained Chunk Management
Storage
Graph Partition Storage

整体执行流程

我们将以伪代码形式展示 GraphCPP 的整体执行流程。该算法接收两个输入参数：当前计算节点所包含的所有图分块的集合 C 以及当前计算节点所包含的所有查询任务的集合 Q。首先，我们分配一个动态大小的连续内存空间，用于存储所有的查询任务（第一行）。然后，我们进入一个循环处理过程，只要仍有未结束的查询任务（第二行），GraphCPP 将调用 ChoseNextSharingChunk 来选择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（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 ci 相关联的所有查询任务（第四行）。接下来，我们将 ci 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 ci 存在关联，将 qi 添加到 Qci，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。

Overall Execution Workflow

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

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:     $c_i$ ← ChoseNextSharingChunk( )
4:     $Q_{ci}$ ← ChoseAssociatedQueries( $c_i$ )
5:     Parallel_for_each $q_i \in Q_{ci}$ do:   // Execute queries in $Q$ in parallel, which is associated with chunk $C$
6:         new_query =GraphCPPCompute( $q_i$, $c_i$ )
7:         if(has Associated( ( $c_i$ , new_query ) ):
8:             $Q_{ci}$.Push( new_query )
9:         else:
10:            $Q$.Push( new_query )
```

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

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

数据访问共享机制

在 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。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。

一、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 C_S 表示要确定的共享的细粒度数据分块的大小，使用 G_S 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$\alpha\times|V|$表示共享分块所拥有的顶点数目的近似值。我们使用 V_S 表示存储一个顶点的状态信息平均所需的空间大小，则$\alpha\times|V|\times V_S$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T_S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。R_S 是预留的冗余空间的大小。LLC_S 是 LLC 缓存空间的大小。则在满足下列不等式的前提下，C_S的最大值就是图分块的大小。

$$\alpha=\frac{C_S}{G_S}$$
$$T_S=\alpha\times|V|\times V_S\times N$$
$$C_S+T_S+R_S\le LLC_S$$

Data Access Sharing Mechanism

In Section 3.2, 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 data. Distributed memory systems use caching to improve data access efficiency. Ideally, the shared graph partition 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 partition should not be too small, as it would increase the synchronization overhead of task processing. The following demonstrates how to determine an appropriate block size by considering both block-level graph structure data and task-specific data.

We use CS to represent the size of the fine-grained data block to be determined for sharing, and GS to represent the size of the graph structure data on each graph partition. α represents the proportion of the shared block part in the partition image. We use |V| to represent the total number of vertices on the partition. $\alpha\times|V|$ represents an approximate value of the number of vertices owned by the shared block. We use VS to represent the average space required to store the status information of a vertex. α×|V|×V_S represents the maximum space required for storing task-specific data on the shared block for query tasks. Considering that multiple cores in multi-core processors execute concurrently, task-specific information for multiple queries needs to be retained in the cache. We use N to represent the number of threads executing parallel computations, and T_S represents the space required to store task-specific data of associated tasks for the current block in the cache. RS is the size of reserved redundant space. LLCS is the size of LLC cache space. Under the premise of satisfying the following inequalities, the maximum value of C_S is the size of the graph block.

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($P_i$ , chunk_set)
2:     chunk_edge_num = 0
3:     chunk = null
4:     for each e ∈ $P_i$ do:   //e is an edge in Partition $P_i$
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 × $\frac{S_G}{|E|}$ ≥ $S_C$:
12:            chunk_set.push(chunk )
13:            chunk_edge_num = 0
13:            chunk.cear( )
14:        end if
15:    end for
```

2. Logical Partitioning. Distributed systems commonly employ partitioning techniques to divide a large-scale graph into graph partitions that can fit into the memory of a single machine. Building upon the memory-level graph partitioning, GraphCPP further divides the graph into fine-grained graph blocks. Unlike previous physical partitioning, the block partitioning here is based on logical divisions. 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 P_i ，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。

Logical Partitioning Function takes two parameters: one is the graph partition structure data Pi recorded in edge table format, and the other is the set of logically divided chunks called chunk_set. Then, define the variable chunk_edge_num to record the current partition's number of edges. Define the variable chunk, which is a dictionary with keys representing source vertex IDs, and values representing the number of outgoing edges for each vertex. Iterate through each edge in the partition. If the edge has already been loaded into the current partition, increment the corresponding count of outgoing edges for that partition. If the vertex is added to the chunk dictionary for the first time, set the count of outgoing edges for the partition to 1. After processing each edge, check if the current chunk is full. If so, add the current chunk to chunk_set. This way, after traversing all the data in the partition, every edge in the partition is assigned to a specific graph block, resulting in a set of logically partitioned graph blocks.

3. Associate Query Tasks with Respective Blocks. In the previous step, we achieved fine-grained block partitioning using a logical approach. Since this partitioning is purely logical and the data remains contiguous on the physical storage medium, it becomes straightforward to determine the block a vertex belongs to based on its ID. Specifically, each query task maintains a record of the active vertex set during the traversal process. Initially, we infer the block in which a vertex resides by examining its ID. Subsequently, we employ a specially designed array to store the partitions traversed by each task. Due to the pruning-based traversal strategy employed in point-to-point queries, the number of active vertices in each round of execution is relatively low. This allows us to establish the association between query tasks and their respective blocks at a low cost.

4. Determining Priority for Partition Scheduling. After establishing the association between query tasks and their corresponding blocks, we can tally the number of tasks associated with each block. A higher task count implies that more tasks share this block, indicating a greater benefit derived from scheduling this block. Consequently, such blocks are given priority during scheduling.

Through the aforementioned steps, we generate partitions for task sharing. By employing an economical priority scheduling sequence, we load these graph partitions into the LLC cache. Subsequently, a fine-grained processing mechanism is required to effectively utilize this shared data.

二、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 q_i 在执行过程中会维护一个活跃顶点集 $Set_{act,i}$，它遵循以下更新策略：1，初始时 $Set_{act,i}$ 仅包含查询源顶点 S_i。2，按照点对点查询算法的流程处理 $Set_{act,i}$ 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 $Set_{act,i}$ 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 q_i 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 $Set_{act,i}$，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。

计算共享机制

全局索引选择高度顶点计算到达所有顶点的距离值，它利用闲时算力维护高度顶点到其余顶点的距离值，从而在不同查询任务中共享这部分高度顶点距离值。但是全局索引机制存在以下缺陷：缺陷 1 全局索引需要记录高度顶点与其它所有顶点的距离值，而当图的规模非常大时，建立索引的计算开销和存储开销会很大。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，所以维护索引的计算开销也很大。

B.Implementing Data Sharing Among Multiple Tasks

Triggering Concurrent Execution of Associated Tasks: Each query task qi maintains an active vertex set Setact,i during its execution, following these update policies: 1. Initially, Setact,i only contains the query source vertex Si. 2. The active vertices in Setact,i are processed according to the workflow of the point-to-point query algorithm. Processed vertices are removed from the active vertex set. 3. If a vertex's state is changed in this round and it is not pruned, the vertex is added to Setact,i, awaiting processing in the next round. In the previous section, we introduced the logical partitioning of graph blocks, where each block corresponds to a chunk dictionary. This dictionary records the IDs of vertices in this block, along with the degrees of vertices in this partition. If the active vertices of task qi appear in the dictionary of a partition, it indicates that this task is an associated task for that partition. Utilizing the chunk dictionary and the active vertex set Setact,i, we can swiftly determine the associated tasks of active blocks loaded into the LLC for concurrent execution. As shown in Algorithm X, after one round of execution for associated tasks, they each generate new active vertices. If these new active vertices remain associated with the current shared block, the query task continues execution. Shared blocks persist in the LLC until all query tasks associated with them have been processed, at which point they are evicted.

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.

一般来说，为了应对随时到来的随机查询，选择的高度顶点的数量越多，对重叠路径的覆盖率越大，计算共享的效果越好。但是基于上面提到的缺陷，我们不能无限制的增加高度顶点的数量，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。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。同时它不再维护高度顶点到所有顶点的距离值，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

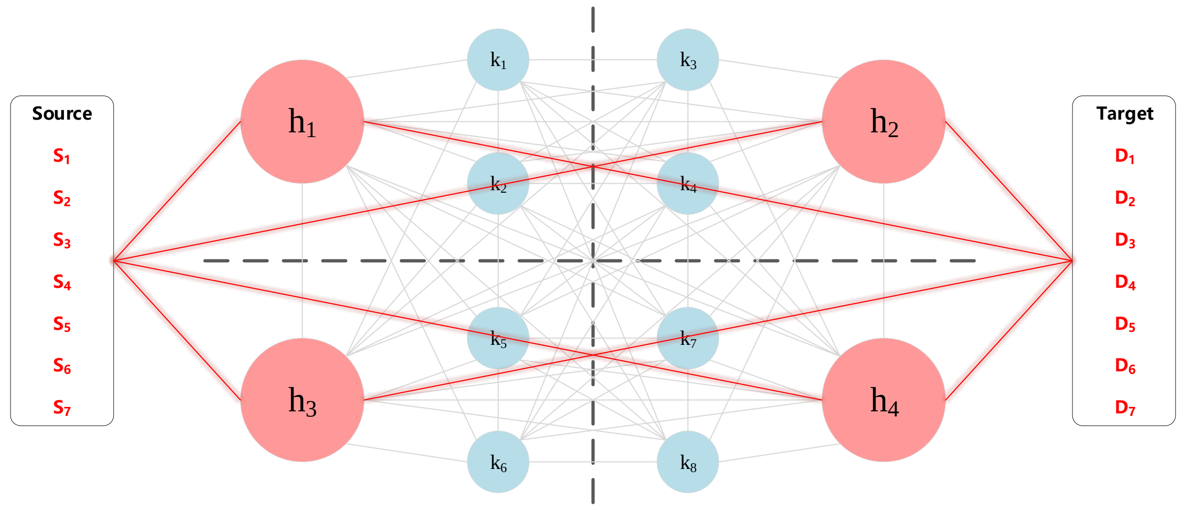
Algorithm: Core-Subgraph Query Acceleration.

```
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
```

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.

实现计算共享的执行步骤如下：1，建立全局索引（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 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，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 1-4): 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 5-8): This step relaxes the degree filtering criteria, allowing more high-degree vertices 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 9-15): 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 16-20): 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.



其它优化

一、维护核心子图

GraphCPP 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”。在计算时，复用全局索引执行基于上界和下界剪枝的点对点查询求得核心子图上高度顶点之间的最短路径值。在存储时，每个高度顶点只需要存储少量的高度顶点之间的距离值。显然，和全局索引相比它的计算开销和存储开销都大大减少。而针对动态图上的索引维护，我们也做了特别优化。具体地，在计算高度顶点之间的最短路径时，每个顶点都会记录其路径上的父节点。当最终路径收敛，从目的顶点逆推可以获得最短路径上所有的顶点的集合，我们将每条最短路径的集合存放在以路径起始点为索引的数组中。当图更新到来，我们首先判断受影响的活跃顶点是否位于某条最短路径上，如果不在，则不会对该条索引产生影响，无需更新，否则需要重新计算该索引的距离值。

Other Optimization

1. Core Subgraph Maintenance:

GraphCPP streamlines the traditional "global index," which maintains distance values for all vertices, into a "core subgraph index" that only maintains distance values between high-degree vertices. During computation, the global index is reused to perform point-to-point queries based on upper and lower bound pruning to obtain the shortest path values between high-degree vertices on the core subgraph. In terms of storage, each high-degree vertex only needs to store a small amount of distance values between high-degree vertices. Clearly, compared to the global index, both the computational and storage costs of the core subgraph index are significantly reduced. Additionally, for index maintenance on dynamic graphs, we have implemented special optimizations. Specifically, when calculating the shortest path between high-degree vertices, each vertex records its parent node on the path. When the final path converges, we can obtain the set of all vertices on the shortest path by retracing from the destination vertex. We store each set of shortest paths indexed by the starting point of the path. When a graph update occurs, we first check whether the affected active vertices are part of any shortest path. If they are not, the index remains unaffected and does not require an update. If they are, we need to recalculate the distance values for that index.

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 Set_S，以及目标顶点的邻居顶点集 Set_D（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 Set_S，目的点位于 Set_D 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.

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。