目录

[动态图上数据驱动的并发点对点查询系统 2](#_Toc149671637)

[摘要 2](#_Toc149671638)

[前言 3](#_Toc149671639)

[背景和动机 6](#_Toc149671640)

[Preliminaries 7](#_Toc149671641)

[并发点对点查询任务的性能瓶颈 8](#_Toc149671642)

[我们的启发 10](#_Toc149671643)

[系统概述 11](#_Toc149671644)

[系统架构 11](#_Toc149671645)

[整体执行流程 12](#_Toc149671647)

[数据访问共享机制 12](#_Toc149671648)

[计算共享机制 17](#_Toc149671649)

[其它优化 20](#_Toc149671651)

[实验评估 21](#_Toc149671652)

[相关工作 22](#_Toc149671653)

[结论 23](#_Toc149671654)

[废弃材料 25](#_Toc149671655)

[素材库： 25](#_Toc149671656)

某些变量用符号表示

待办：完善实验统计，补足图和数据部分。

数据驱动的并发点对点查询系统

摘要

随着图处理技术在地图导航、网络分析等领域的大范围应用，大量点对点查询任务在同一个底层图上并发运行，对图查询系统的吞吐量提出了极高的要求。然而已有的图计算系统要么聚焦于优化单次点对点查询的速度，而忽略了并发点对点查询的吞吐量。要么沿袭点对多点算法的并发执行思路，忽略了点对点算法的优化潜力。由于冗余的数据访问开销和计算开销，现有方案在执行并发点对点查询时整体吞吐量很差。

本文提出了GraphCPP，它是第一个并发执行点对点查询任务的图遍历系统，通过数据访问共享和热路径计算共享提高并发查询任务的吞吐量。GraphCPP有两个创新之处。首先，基于不同查询任务的遍历路径大量重叠的观察，GraphCPP提出了一个数据驱动的缓存执行机制，通过细粒度的图分块调度，实现了并发任务之间的**数据共享**，提高了数据访问效率； 其次，由于不同任务对相同的热顶点和热路径频繁计算，GraphCPP提出了一个双层级计算共享机制，它通过对热顶点和热路径的计算共享来加速未知查询的收敛。为了展示GraphCPP的效率，我们在xx等数据集上，将其与最先进的点对点查询系统和并发图计算系统进行对比。结果表明，GraphCPP仅产生xx的预处理开销和xx的存储开销，但是整体吞吐量相较于SGraph、Tripoline、Pnp、Glign，分别提高了xx-xx倍。

（无需翻译：参考了GraphM和OSDI22文章）

索引关键词: graph process, point-to-point graph queries, concurrent jobs, data access similarity，computational similarity

GraphCPP: A Data-Driven Graph Processing System for Concurrent Point-to-Point Queries

Abstract

With the widespread adoption of graph processing technology in fields such as map navigation and network analysis, a considerable number of point-to-point query tasks run concurrently on the same underlying graph, imposing a high demand on the throughput of graph query systems. However, existing graph computing systems either focus on optimizing the speed of individual point-to-point queries, neglecting the throughput of concurrent point-to-point queries, or follow the concurrent execution approach of point-to-multipoint algorithms, overlooking the optimization potential of point-to-point algorithms. Due to redundant data access overhead and computational expenses, existing solutions exhibit poor overall throughput when executing concurrent point-to-point queries.

This paper introduces GraphCPP, the first graph traversal system designed for concurrent execution of point-to-point query tasks. It enhances the throughput of concurrent query tasks through data access sharing and hot path computation sharing. GraphCPP is novel in two ways. Firstly, based on the observation that traversal paths for different query tasks overlap significantly, GraphCPP proposes a data-driven caching execution mechanism. Through fine-grained graph chunk scheduling, this mechanism enables data sharing among concurrent tasks, thereby enhancing data access efficiency. Secondly, recognizing the frequent computation of the same hot vertices and paths by distinct tasks, GraphCPP proposes a dual-level computation sharing mechanism. This mechanism accelerates the convergence of unknown queries by sharing computed values of hot vertices and paths. In our evaluation on datasets such as xx, we compare GraphCPP with state-of-the-art point-to-point query systems and concurrent graph processingsystems. The results demonstrate that GraphCPP incurs only xx preprocessing overhead and xx storage overhead, achieving a throughput improvement of xx-xx times compared to SGraph, Tripoline, Pnp, and Glign.

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

前言

大量并发点对点查询任务通常在同一底层图上执行，如：在物流运输时，谷歌地图[xx]会找到两个地点之间的最佳路径；在社交网络分析时，Facebook[x]会通过查找两个用户之间的关系链，为用户推荐可能的朋友；在金融风险分析时，支付宝[x]会分析风险是如何从一个实体传播到另一个实体；这些热门应用提出了在同一个底层图上执行大规模并发点对点查询的需求。然而已有的点对点查询的解决方案[xxxx]聚焦于加速单次查询的效率，而忽略了对并发查询的优化。为了实现并发点对点查询任务的高效执行，需要解决两个关键的挑战。

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

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

参考graphfly）

其次，除了重复访问外，不同查询任务还需要重复计算热门路径的距离值。由于图数据往往具有幂律分布的特点，少量热门顶点会频繁出现在不同查询的最佳路径中。由于不同查询任务之间不会共享计算，它们会重复计算连接热顶点的热路径。此外，热门顶点往往是具有大量邻居顶点的高度顶点，对于它们的重复遍历常常导致计算开销的爆炸式增长。一些已有的系统中尝试采用全局索引[xxxx]的方式来实现计算共享，但由于昂贵的计算开销、存储开销、更新开销限制了计算共享的效率和精确度。

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

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

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

在GraphCPP中，我们提出了一种数据驱动的缓存执行机制，将传统的“任务→数据”的调度方式改为“数据→任务”的调度方式，进而实现多任务之间重叠图结构数据访问的共享。在这种执行机制下，GraphCPP会首先确定数据的调度顺序：它将图结构数据从逻辑上划分为LLC级别的细粒度分块。接着根据查询任务活跃顶点集所处的图分块将查询任务与其相关的图分块联系起来，关联任务数量越多的分块优先级越高更可能被优先调度。任务的活跃顶点每轮都会变化，共享分块的关联任务数也需要每轮更新；为了实现“数据→任务”的调度方式，GraphCPP采用了一种关联任务触发机制。在按照优先级顺序将图分块加载到LLC后，利用每一轮统计得到的任务与数据分块的关联信息，触发当前当前分块的关联任务批量执行，实现了对共享数据的高效访问；

其次，GraphCPP提出了一个包含全局索引和核心子图索引的双层级的计算共享机制。它将图中具有大量连接边的顶点称为热顶点，将热顶点之间的路径称为热路径。虽然热顶点和热路径的数量较少，但是在不同查询任务到的遍历路径中频繁出现。双层级的计算共享机制通过全局索引实现第一层计算共享，全局索引维护了少量热顶点到其它顶点的索引信息，这些热顶点充当了不同查询路径的中介节点，通过它可以为绝大多数查询提供可供剪枝的路径值。通过核心子图机制，我们实现了第二层级的计算共享。核心子图主要维护图中连接热顶点的热路径。注意，核心子图中热顶点数比全局索引中的热顶点数大一个数量级。具体地，它首先要从原始图数据中筛选高度数的热顶点，接着遍历这些顶点之间的热路径，将热路径的路径值作为热顶点对的边的权重。在查询过程中，核心子图就像一个高速公路网，借助它可以从一个热顶点出发遍历到其他热顶点的过程，相当于从该热顶点进入高速公路，借助高速公路网上的快速通道（热路径），快速到达其他高速站点（热顶点）。GraphCPP两层级的计算共享机制减少了冗余计算。

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

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

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

Besides, GraphCPP proposes a dual-level computation sharing mechanism comprising global indexing and core subgraph indexing. It designates vertices with a substantial number of connecting edges in the graph as hot vertices and the paths between hot vertices as hot paths. Although the proportion of hot vertices and hot paths is small, they occur frequently in traversal paths to different query tasks. The dual-level computation sharing mechanism achieves the first level of computation sharing through global indexing. The global index maintains a small set of hot vertices and their index information to other vertices. These hot vertices act as intermediary nodes for different query paths, providing prunable path values for the majority of queries. Through the core subgraph mechanism, the second level of computation sharing is realized. The core subgraph primarily maintains hot paths connecting hot vertices in the graph. It is essential to note that the number of hot vertices in the core subgraph is an order of magnitude larger than that in the global index. Specifically, it first filters hot vertices with high degrees from the original graph data. Subsequently, it traverses hot paths between these vertices, assigning the path values as weights to the edges between pairs of hot vertices. During the querying process, the core subgraph plays a role similar to that of a high-speed highway network. Utilizing it allows traversal from one hot vertex to another, akin to entering a highway at a hot vertex and swiftly reaching other high-speed stations (hot vertices) through the fast lanes (hot paths) on the highway network. The dual-level computation sharing mechanism in GraphCPP mitigates redundant computations.

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

本文主要做出了如下贡献：

1. 分析了现有点对点查询系统处理并发点对点查询任务时的性能瓶颈，并提出利用并发查询任务之间的数据访问相似性和计算相似性来提高并发任务吞吐量。
2. 开发了GraphCPP，一个动态图上数据驱动的并发处理点对点查询系统，实现了并发任务之间的数据共享和计算共享，并提出了一个相似任务批量执行策略。
3. 我们将GraphCPP与当前最先进的三个点对点查询系统XXXXXX进行对比，该工作负载包x个真实图表上的x个应用程序xxxxx。我们的实验表明, 与其他 GPS 相比，GraphCPP平均比xxx、xxx和xxx 提高了 xx倍、xxx 倍和 xxx 倍

This paper makes the following contributions:

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

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

We compare GraphCPP with three state-of-the-art point-to-point query systems, namely XXXXXX, using a workload that includes x real-world graphs and x applications. Our experiments demonstrate that, on average, GraphCPP outperforms others, achieving xx times, xxx times, and xxx times improvement compared to XXX, XXX, and XXX, respectively.

背景和动机

**Table 1: Profiling performance analysis of conducting 10,000 point-to-point queries on LiveJournal graph using existing graph query systems.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| System | Pnp | | | Tripoline | | | SGraph | | |
| Number of concurrent query tasks | 1 | 128 | 512 | 1 | 128 | 512 | 1 | 218 | 512 |
| Instructions (×10^14) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| LLC loads (×10^12) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| LLC miss ratio | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Runtime (hour) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

大多数现有的解决方案[xxxx]聚焦于加速单次查询的速度，然而实际应用场景中，有大量图查询任务在同一个底层图上并发运行。例如，中国地理信息产业协会[x]的统计表明，百度地图[xx]、高德地图[xx]、腾讯地图[xx]、华为地图[xx]等企业建设的位置服务开放平台日均位置服务请求次数最高达1600亿次[xx]。大量并发点对点查询需求对图遍历系统的吞吐量提出了极高的要求。然而如表1所示所示，我们证明现有系统在处理大规模并发查询时吞吐量很差。这种坏结果出现的原因是并发任务之间存在对图结构数据大量的冗余访问和冗余计算。为了定性地分析上述问题，我们在XXXXX（机器配置），选取了XXXXX（现有最佳方案），在XXXXX（图数据集上），进行并行点对点查询的性能评测。

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

BACKGROUND AND MOTIVATION

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

Preliminaries

定义一：（图）我们使用G=(V,E)来表示有向图，其中V是顶点的集合，E是由V中顶点组成的有向边的集合（无向图中的边可以被拆分为两个不同方向上的有向边）。我们使用|V|，|E|分别表示顶点的数目以及边的数目。

定义二：（图分区）我们使用Pi=(VPi,EPi)来表示有向图的第i个图分区，使用VPi表示图分区中顶点的集合，EPi是由VPi中顶点组成的有向边的集合。对于分布式系统，不同机器上的图分区Pi各不相同，我们采用边切分的方式划分图，同一个顶点可能出现在不同计算节点上，但是只有一个主顶点，其它的都是镜像顶点。

定义三：（点对点查询）我们使用qi=(si，di)表示任务i对应的查询任务。其中si和di分别表示查询qi对应的源顶点和目的顶点。假如以为si和di输入，在图上执行点对点查询算法，可以得到一条收敛的路径，它就是si和di之间的最佳路径。

定义四：（界限）主流的点对点查询系统中，普遍采用了基于剪枝的查询策略， 界限提供了保守的剪枝值。具体地，界限可进一步划分为上界和下界，上界UB表示当前已知的从源点到目的顶点的最佳路径的路径值，下界则LB表示从当前顶点v到目的顶点保守的最佳距离预测值，预测的LB小于或等于顶点v到目的顶点实际的最佳距离。根据图上的三角不等式，如果一条路径的距离大于UB，或者加上LB的值后比UB大，则这条路径一定比已有的路径差，需要被剪枝。上下界的值需要借助索引来推导出，它们本质上是一种计算共享。

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

**Preliminaries**

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

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

Definition 3: (Point-to-Point Query) We use qi=(si, di) to denote the query corresponding to task i, where si and di represent the source and destination vertices of query qi, respectively. If the execution of the point-to-point query algorithm on the graph yields a convergent path for qi, this path is referred to as the optimal path between si and di.

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

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

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| **图1** | 图2 | | 图3 |
| 图4 | 图5 | 图6 | |

并发点对点查询任务的性能瓶颈

在本节，我们实现了现有线性点对点查询系统的并发版本以及现有并发图计算系统的点对点版本，以定量地说明现有并发点对点查询方案的性能瓶颈。

**并发任务的****冗余数据访问开销。**我们观察到当并发点对点查询任务在同一个底层图上执行图遍历时，体现出来很明显的数据访问相似性。如图1，并发任务之间存在很大比例的访问重叠，特别地，相似任务的重叠比例更高(more details are given in Sectionxx)。然而在传统的“任务->数据”调度模式下，不同任务独立执行查询，因此它们会相互争用有限的缓存来存储各自所需要的图数据分块，哪怕这些数据分块是完全相同的。这会导致严重的缓存抖动并降低处理并发点对点查询的性能。表1表明，随着并发数目的增大，查询任务的缓存未命中率也急剧增大。并且虽然并发执行的整体效果优于线性执行，但是由于前面提到的冗余数据访问开销，随着查询任务数量额增加每个任务的平均查询时间显著增加。注意，并发方式的总执行时间是这些作业执行时间的最大值，而顺序方式是所有作业执行时间的总和。

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

In this section, we have implemented the concurrent version of the existing linear pairwise query system and the pairwise version of the existing concurrent graph computing system. Our objective is to quantitatively elucidate the performance bottlenecks of current concurrent pairwise query solutions.

The redundant data access overhead of concurrent tasks is evident. We observe that when concurrent pairwise query tasks execute graph traversal on the same underlying graph, a significant portion of their traversal paths exhibits clear data access similarities. As shown in Figure 3x, our data indicates that there is a substantial overlap in data access among concurrent tasks, with a higher overlap ratio for similar tasks (more details are provided in Section xx). However, in the traditional "task->data" scheduling mode, different tasks independently execute queries, leading to competition for limited cache space to store their respective graph data chunks, even when these data chunks overlap. This results in severe cache thrashing and a reduction in the performance of processing concurrent pairwise queries. Table 1 illustrates that, with an increase in the number of concurrent tasks, the cache miss rate for query tasks sharply rises. Although the overall performance of concurrent execution surpasses linear execution, the average query time per task significantly increases with the growing number of query tasks due to the mentioned redundant data access overhead. It is essential to note that the total execution time in the concurrent mode is the maximum among the execution times of these tasks, while in the sequential mode, it is the sum of all task execution times.

**并发任务的冗余计算开销。**由于图数据的幂律分布特性，少数热顶点连接了大部分边。因此，如图2所示，尽管热顶点仅占总顶点数的一小部分（XX%），它们却出现在许多路径中（XX%）。并且共享程度越高的图数据中热顶点的比例越高。这意味着不同的查询任务会重复计算热顶点之间的热路径，体现出了并发任务之间的计算相似性。在一个图快照周期内，对相同顶点对的路径计算结果是相同的，这意味着并发任务的大量计算是冗余的。此外，由于热顶点往往拥有大量出边和入边，它带来的计算开销也远大于普通的顶点。

一些现有的解决方案尝试建立全局索引机制[x]来利用计算相似性。然而全局索引机制存在天然缺陷。缺陷1：全局索引需要记录热顶点与普通顶点的路径值，当图的规模非常大时，建立索引的计算开销和存储开销会很大。如图3，4所示，全局索引的计算开销和存储开销，随着全局索引顶点数的增加而等比例增加。缺陷2：在动态图处理中，每轮图更新都会导致新的边添加和边删除产生。现有的方案采用了增量更新的方式更新全局索引。假如受影响的顶点恰好处于某一条最佳路径上，那么从受直接影响的顶点到终点这段路径上的所有顶点的全局索引值都需要重新计算。假如共享的数据部分发生了图更新，其影响会进一步扩大。如图5所示，随着图更新比例增加，需要重新计算的全局索引的比例不断增加。缺陷3：不同图数据集的数据分布情况也不同，很难选择一个合适的索引顶点数目以在索引的覆盖范围（索引覆盖范围越大，计算共享的有效率越高）和索引本身的开销之间取得平衡。如图6所示，在不同的数据集上选取同样的索引顶点数，其固有开销和计算共享的有效性变化很大。综上，全局索引机制本身会带来昂贵地开销，且针对不同的数据集所需开销的波动很大。因此现有的系统往往保守地选取全局索引的数目（在Tripoline和SGraph中都选择16个顶点），以避免带来昂贵地开销，但这样也限制了全局索引的覆盖范围，无法实现高效地计算共享。

Redundant Computational Overhead of Concurrent Tasks. Due to the power-law distribution characteristics of graph data, a small number of hot vertices are connected to the majority of edges. Therefore, as shown in Figure 2, although hot vertices constitute only a small portion of the total vertices (XX%), they appear in many paths (XX%). The proportion of hot vertices in graph partitions with higher sharing levels is even higher. This implies that different query tasks will redundantly calculate hot paths between hot vertices, demonstrating computational similarity among concurrent tasks. Within a graph snapshot period, the results of path calculations for the same vertex pairs are identical, indicating that a significant amount of computation in concurrent tasks is redundant. Additionally, as illustrated in Figures 3 and 4, due to the fact that hot vertices often have a large number of outgoing and incoming edges, the computational cost they bring is much greater than that of ordinary vertices.

Some existing solutions attempt to establish a global indexing mechanism [x] to leverage computational similarity. However, the global indexing mechanism has inherent flaws. Flaw 1: The global index necessitates recording path values between hot vertices and regular vertices. When the graph is of considerable scale, the computational and storage costs of establishing the index can be substantial., as shown in Figures 3 and 4, where the computational and storage costs of global indexing increase proportionally with the number of global index vertices. Flaw 2: In point-to-point queries on streaming graphs, each round of graph updates introduces new edges and edge deletions. The global index needs to dynamically update the index relationships between high-degree vertices and every vertex based on the latest graph snapshot. This implies that any update to the streaming graph will impact all vertex indices, as illustrated in Figure 5, where a small number of graph updates lead to extensive updates in the global index. Flaw 3: Different datasets have different data distribution patterns, making it challenging to choose an appropriate number of index vertices to balance the efficiency of computational sharing and the overhead of the index itself. As shown in Figure 6, selecting the same number of index vertices on different datasets results in significant variations in inherent costs and the effectiveness of computational sharing. In summary, the global indexing mechanism itself incurs expensive costs, and the fluctuation in required costs for different datasets is considerable. Therefore, existing systems often conservatively choose the number of global indices (e.g., 16 vertices in Tripoline and SGraph) to avoid incurring excessive costs. However, this also limits the coverage range of global indices, preventing efficient computational sharing.

我们的启发

通过上述的观察，我们得到了以下启发：

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

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

**Our Motivation**

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

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

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

系统概述

系统架构

为了提高并发点对点查询的执行效率，在对并发点对点查询的计算细节进行仔细研究后，我们提出了一个数据驱动的高效并发点对点查询系统-GraphCPP。我们系统的关键思想是实现并发点对点查询任务之间的数据共享和计算共享。如下图所示，为了实现这个目标，GraphCPP包含了一个高效地数据驱动的缓存执行机制，利用并发任务之间的数据相似性，实现了重叠数据的共享访问。以及一个双层级计算共享机制，共享不同查询任务对相同热顶点和热路径的计算。此外，它还通过预测不同查询的遍历路径，驱动路径重叠的相似查询批量执行，进一步利用了数据相似性。

数据访问共享机制。它负责利用并发任务的数据访问相似性实现图结构数据的细粒度共享。首先，它像其它分布式图计算系统[13-16]一样将原始图数据划分为粗粒度的图分区，交由不同的机器并行处理。然后使用一个细粒度的分块管理器将粗粒度的图分区划分为细粒度的图分块。接着执行图分块-任务之间的关联机制：假如一个查询任务qi在某个图分块bi存在活跃顶点，那么就认为qi和bi之间存在关联关系。分块优先级调度机制会根据不同图分块包含关联任务的数量，优先调度有更多关联任务的图分块到LLC。关联任务触发器会根据LLC中活跃图分块信息，以及图分块-任务的关联信息，筛选出所有有关联关系的任务在共享图分块上批量执行。

计算共享机制。它负责借助双层级索引信息（包含全局索引和核心子图索引）利用并发查询任务之间的计算相似性实现路径计算的共享。1）**预处理阶段**：它首先统计原始图数据中所有顶点的度数信息， 将原始图的顶点按照度数由大到小排序，选择度数排名位于[1,k]的顶点作为全局索引顶点，选择度数排名位于[k+1,k+m]的顶点作为核心子图索引顶点；2）**索引建立阶段**：一方面，在查询开启前执行点对多点查询算法计算全局索引顶点与图中所有顶点的最佳路径值（例如对于PPSP任务，需要执行SSSP算法），对于有向图，需要分别计算全局索引的出边路径值和入边路径值，对于无向图则只需要计算一次；GraphCPP Overview

**system architecture**

In order to enhance the execution efficiency of concurrent point-to-point queries, we propose a data-driven and efficient concurrent point-to-point query system, GraphCPP, after a meticulous examination of the computational intricacies of concurrent point-to-point queries. The fundamental concept of our system is to achieve data and computation sharing among concurrent point-to-point query tasks. As depicted in the figure below, GraphCPP incorporates an efficient data-driven caching execution mechanism. It exploits data similarity among concurrent tasks to facilitate shared access to overlapping data. Additionally, it introduces a dual-level computation sharing mechanism based on the core subgraph, enabling the sharing of computations for the same hot paths across different query tasks. Moreover, it employs path prediction for different queries, driving the bulk execution of similar queries with overlapping paths and further leveraging data similarity.

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

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

另一方面：GraphCPP通过一个运行时方法来动态维护核心子图。具体地，一个顶点对的最佳路径的任意一段子路径，都是对应顶点之间的最佳路径。因此核心子图无需对热路径进行预计算，而是在每次查询结束后，执行热路径探测机制发掘一条最佳路径中的热路径，并将其加入到核心子图结构中。3）计算共享阶段：全局索引顶点一般充当大量路径的中介枢纽顶点，因此在查询开始时，使用全局索引顶点可以计算出一条可达路径的路径值。这条路径不一定是查询顶点对之间的最佳路径，但是可以对剪枝查询提供参考。此外在基于上界+下界的剪枝遍历中，还可以借助全局索引预估查询路径的路径值，更早对不合适路径进行剪枝。使用全局索引是第一层次的计算共享；核心子图维护了热顶点之间的最佳路径值，它充当了查询任务之间的高速公路。当查询任务遍历到核心子图中的热顶点，相当于接入了高速公路，它可以借助核心子图从入口顶点快速到达核心子图的出口顶点，而无需重复计算这段热路径的路径值。使用核心子图是第二层次的计算共享。

整体执行流程

算法1展示了GraphCPP处理并发查询任务的整体执行流程。假设我们已经完成分区，在第一行我们得到在分布式系统中属于当前计算节点的所有图分块的集合B以及所有查询任务的集合Q，并为他们分配一段内存空间。在第二行，我们进入一个循环处理过程，查询会迭代进行直至收敛。GraphCPP将调用ChooseNextSharingBlock来更新查询任务与图分块之间的关联关系，并选择当前优先级最高(有着最多关联任务)的图分块bi。通过统计每个任务的关联分块（即任务在当前分块存在活跃顶点），我们可以确定与当前图分块bi相关联的所有查询任务（第四行）。接下来，我们将bi加载到缓存，并并行处理所有相关联的查询操作qi（第五行）,这一步体现了数据共享的思想，即一次数据访问，多个查询任务共享。接着我们调用GraphCPPCompute在当前分块上执行点对点查询操作qi（Details are given in section.xx）。迭代查询任务执行BSP模型，每一轮迭代会生成新的活跃顶点集，并更新为一个新的查询任务（第六行）。如果新查询依然与当前的图分块bi存在关联，将qi添加到Qbi，然后返回第五行继续在图分块bi上查询。否则，代表新任务与当前图分块没有关联关系，需要将其保存在查询任务集合中，此时任务被挂起。

|  |
| --- |
| Algorithm 1: Concurrent Point-to-Point Queries |
| 1: function OverallWorkflow  2 MallocBuffers( *B*, *Q* ) //*B* is the set of graph blocks, and *Q* is the set of query tasks  3: While has\_active( *B* ) do：  4: *bi* ← ChooseNextSharingBlock( )  5: *Qbi* ← ChooseAssociatedQueries( *bi* )  6: Parallel\_for\_each *qi* ∈ *Qbi* do: // Execute queries in *Q* in parallel, which is associated with block *bi*  7: new\_query =GraphCPPCompute( *qi*, *bi* ) // The implementation function for point-to-point queries returns the active vertex set after one round of task iteration.  8: if(has Associated( ( *bi* , new\_query ) ):  9: *Qbi*.Push( new\_query )  10: else:  11: *Q*.Push( new\_query ) |
|  |

Algorithm 1 outlines the comprehensive execution process of GraphCPP in handling concurrent query tasks. Assuming partitioning is completed, in the first line, we acquire the set \(B\), which includes all graph blocks on the current computing node, and the set \(Q\), which includes all query tasks on the current computing node. Each of them is allocated a segment of memory. In the second line, we enter a loop processing phase, and queries iterate until convergence is achieved. GraphCPP invokes \texttt{ChooseNextSharingBlock} to update the association between query tasks and graph blocks and selects the graph block \(b\_i\) with the highest priority (having the most associated tasks). By tallying the associated blocks for each task (i.e., tasks with active vertices in the current block), we can determine all query tasks related to the current graph block \(b\_i\) (fourth line). Next, we load \(b\_i\) into the cache and parallelly process all associated query operations \(q\_i\) (fifth line). This step embodies the idea of data sharing, i.e., multiple query tasks sharing the result of a single data access. Subsequently, we invoke \texttt{GraphCPPCompute} to perform point-to-point query operations \(q\_i\) on the current block (details are provided in Section \ref{section:xx}). Iterative query task execution follows the Bulk Synchronous Parallel (BSP) model, where each iteration generates a new set of active vertices and updates to form a new query task (sixth line). If the new query remains associated with the current graph block \(b\_i\), we add \(q\_i\) to \(Q\_{b\_i}\) and return to the fifth line to continue querying on the graph block \(b\_i\). Otherwise, it signifies that the new task is not associated with the current graph block, and it needs to be saved in the query task set; at this point, the task is suspended.

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

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

数据访问共享机制

在2.2节中我们观察到并发任务之间的图结构数据访问存在很大一部分重叠，在现有处理机制下，这部分重叠数据并不能被共享利用，我们观察到，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，并基于此提出了数据共享机制。该机制的核心思想是将原本的“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，来利用并发查询任务之间的数据相似性，分摊数据访问开销，提高缓存利用效率，进而提高系统吞吐量。接下来我们从两个问题出发描述如何实现数据访问共享，并在最后描述了一个进一步利用数据访问相似性的措施。

一、如何确定共享的数据部分？

1，确定共享图分块粒度。分布式内存图计算系统需要将数据载入缓存以提升数据访问效率，所以理想情况下共享图分块的数据要能完整地载入LLC，从而避免访问分块不同部分带来的频繁换入换出。但是图分块的粒度也不能过于小，否则会增加任务处理的同步开销。我们使用公式x来确定合适的共享图分块大小。其中BS表示待确定的共享图分块的图结构数据的大小，GS表示分块所属的图分区的图结构数据的大小，|V|表示分区上图的顶点总数，VS表示存储一个顶点的状态信息平均所需的空间大小，N表示并发查询的任务数，LLCS是LLC缓存空间的大小，RS是预留的冗余空间的大小。公式x右侧的两项分别表示图结构数据和任务特定数据（其大小与图分块的规模和并发查询任务数成正比）。公式右侧表示减去缓存预留空间后，每个任务剩下的可使用空间的大小。通过这个公式，我们求得了在适应LLC容量前提下，每个共享图分块的最大粒度。

**Data Access Sharing Mechanism**

In Section 2.2, we observed a significant overlap in 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. The core idea of our data sharing mechanism is to transform the original "task → data" linear task scheduling order into a "data → task" fine-grained concurrent task scheduling order. This allows us to leverage data similarity among concurrent query tasks, distribute data access overhead, enhance cache utilization efficiency, and consequently, improve system throughput. We will address two key questions in the following sections: 1) How to determine the shared data segments? 2) How to implement data sharing among multiple tasks? Finally, we will describe additional measures to further exploit data access similarity.

A. How to Determine Shared Data Segments?

1.Determine the granularity of shared graph block. Distributed memory graph computing systems need to load data into the cache to improve data access efficiency. Ideally, the data of shared graph should be able to fit entirely into the Last Level Cache (LLC), thereby avoiding the frequent swapping in and out of block parts. However, the granularity of graph blocks should not be too small, as it would increase the synchronization overhead of task processing. We employ formula x to determine an appropriate size for shared graph blocks. In this formula, BS represents the size of the graph structure data for the to-be-determined shared graph block, GS denotes the size of the graph structure data for the partition to which the block belongs, signifies the total number of vertices on the partition, VS stands for the average space needed to store the status information of a vertex, N represents the number of concurrently queried tasks, LLCS denotes the size of the LLC cache space, and RS refers to the reserved redundant space. The two terms on the right side of formula x respectively represent the graph structure data and task-specific data (whose size is proportional to the scale of the graph block and the number of concurrently queried tasks). The right side of the formula indicates the size of the available space for each task after deducting the reserved cache space. Through this formula, we determine the maximum granularity of each shared graph block under the condition of accommodating the LLC capacity.

2，逻辑划分。确定好共享图分块的粒度后，GraphCPP就可以在图预处理的过程中，采用逻辑划分的方式，将分布式系统上粗粒度的图分区划分为细粒度的共享图分块。代码2展示了GraphCPP划分图分块的过程：

|  |
| --- |
| Algorithm 2: Logical Partition Algorithm. |
| 1: function Partition(Pi , block\_table) // block\_table records the set of graph blocks owned by graph partition Pi.  3: block\_map = null  4: for each e ∈ Pi do: //e is an edge in partition Pi  5: if e.src in block\_map:  6: block\_map [e.src]++  7: else:  8: block\_map[e.src]=1  9: end if  11: if block\_map.size（）≥ SC:  12: block\_table.push(block\_map)  13: block\_map.cear( )  14: end if  15: end for |

2. Logical Partitioning. Once the granularity of shared graph blocks is established, GraphCPP can proceed with the logical partitioning during the graph preprocessing phase. This process involves subdividing coarse-grained graph partitions on the distributed system into finer-grained shared graph blocks. Pseudocode for partitioning graph blocks in GraphCPP is presented in Listing X:

逻辑分区函数接收两个参数，一个是以边表形式记录的图分区结构数据Pi ，一个是该分区所拥有的图分块集合block\_table（第一行）。在第二行，我们通过一个字典结构block\_map来统计图分块信息，它的key记录边的源顶点ID，value记录该顶点对应的出边的数目。第四行，GraphCPP循环遍历分区中的每一条边。如果该边已经被加载到当前的分区，将分区对应的出边数量加一(第6行)。如果该顶点是第一次加入到字典中，将分区的出边数置为1（第8行）。每次遍历完一条边都会判断当前分块是否已满（第11行），若分块已满，将当前分块加入图分块集合block\_table（第12行），并清空记录的分块信息（第13行）。这样当分区中的所有数据遍历完一遍，分区的每一条边都被划归到某一个图分块，我们就得到了从逻辑上划分完成的图分块的集合。

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

二、如何实现多任务间的数据共享

建立共享分块-查询任务的关联。通过之前的步骤，我们用逻辑划分的方式，实现了细粒度的图分块。我们使用block\_table来维护逻辑分块信息，其中每个分块附加上了一个标签信息。执行查询时，每个任务qi在迭代计算过程中会维护一个活跃顶点集Setact,i，它遵循以下更新策略：1，初始时Setact,i仅包含查询源顶点Si。2，按照点对点查询算法的流程，从Setact,i中的活跃顶点出发查询新的顶点，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。通过上述更新策略，我们可以轻松追踪每一轮迭代各个任务的活跃顶点集。接下来借助block\_table可以查询活跃顶点所在的图分块。进一步地，我们可以统计每个图分块有哪些关联任务。由于点对点查询采用基于剪枝的遍历策略，每一轮执行中活跃顶点的数量并不多，所以可以以较低的开销建立查询任务与所属分块的关联。

2，确定分区调度的优先级。对共享图分块的访问开销可以被分摊到多个任务中，因此分块的关联的任务数量越多，共享带来的收益越大。由于在上一步已经获得了共享分块和查询任务的映射关系，因此通过遍历block\_table，我们可以快速获得每个图分块的优先级。

3，触发关联任务并发执行。确定图分块优先级后，我们优选高优先级的共享分块加载到LLC中，接着根据共享分块-查询任务的关联关系推导出所有在当前图分块活跃的查询任务，并采用批量计算的方式执行这些任务。如算法X所示，活跃任务执行一轮后会产生新的活跃顶点，倘若新的活跃顶点仍然与当前的共享分块相关联，查询任务会继续执行。共享分块会始终停留在LLC，直到与该分块关联的所有查询任务都被处理完毕，才会换出。

B. Achieving Data Sharing Among Multiple Tasks

Establishing the Association between Shared Blocks and Query Tasks. Through the previous steps, we achieved fine-grained graph partitioning in a logical manner. We use the \texttt{block\_table} to maintain information about logical blocks, with each block appended with a label. As this is only a logical partitioning, the data remains contiguous on the physical storage medium. Therefore, it is easy to determine the partition in which a vertex is located based on its ID. During query execution, each task $q\_i$ maintains an active vertex set $Set\_{act,i}$ throughout the iterative computation process. It follows the updating strategy:

\begin{enumerate}

\item Initially, $Set\_{act,i}$ only contains the query source vertex $S\_i$.

\item Process the active vertices in $Set\_{act,i}$ according to the flow of the point-to-point query algorithm, removing the processed vertices from the active set.

\item If a vertex's state changes in this round and it is not pruned, add the vertex to $Set\_{act,i}$ for processing in the next round.

\end{enumerate}

With the above updating strategy, we can easily track the active vertex sets for each task in each iteration. Subsequently, leveraging the \texttt{block\_table}, we can first query the ID of active vertices and deduce the graph block to which they belong. Furthermore, we can then tally and utilize a specially designed array to store which partitions each graph block is associated with for each task. Since point-to-point queries adopt a pruning-based traversal strategy, the number of active vertices in each execution round is not high, allowing for the establishment of the association between query tasks and their respective blocks with relatively low overhead.

Determining the Priority of Partition Scheduling. After establishing the association between query tasks and their corresponding blocks, we can tally the number of tasks associated with each graph block. The higher the task count, the more tasks share the block, indicating greater benefits from processing this block. Consequently, blocks with a higher task count are prioritized for placement into the Last-Level Cache (LLC). Since we obtained the mapping relationship between shared blocks and query tasks in the previous step, by traversing the \texttt{block\_table}, we can quickly obtain the priority of each graph block.

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

三、相似任务批量执行

同一时刻并发任务池中有大量随机查询任务，它们的最佳查询路径大不相同。我们发现其中一些任务的相似程度很低，即它们的查询任务重叠程度很低，甚至可能没有重叠。与之相应的另一个极端，一些任务的相似程度很高，例如查询A的路径包含了查询B的路径，对于查询B来说，它的路径完全和查询A重叠。对于我们的数据共享机制，路径的重叠率越高，数据的访问效率越高，冗余的同步迭代更少。对此我们提出了一个相似任务感知的批量执行策略，每次从任务池中筛选相似任务批量执行，以进一步地利用数据相似性和计算相似性。具体地，GraphCPP首先从任务池中随机选择一个查询任务，获取任务的起始顶点和目标顶点。然后执行BFS算法获取起始顶点的邻居顶点集SetS，以及目标顶点的邻居顶点集SetD。注意，考虑到部分中心节点存在大量邻居节点，我们设置邻居节点的上限为500。随后遍历任务池，筛选出所有起始点位于SetS，目的点位于SetD的查询任务，它们被作为相似任务并发处理。需要注意的是，如果某个查询的起始顶点或目的顶点属于高度顶点，可以直接使用索引来加速查询过程，无需使用常规的查询步骤。排除掉高度顶点后K跳SSSP本身的开销很小，且执行过程可以和正常查询并发执行，执行开销可以忽略不计。

计算共享机制

GraphCPP通过全局索引机制和核心子图索引机制实现了两个层次的计算共享。全局索引的固有开销很大，因此实践中全局索引顶点的数目设置的很少（通常为16）。然而由于幂律分布的特点，这些热顶点充当了不同查询的中介枢纽节点，因此对于大部分点对点查询，都可以找到一条途径源顶点、全局索引顶点和目的顶点的路径。虽然很难确保途径这条路径一定是查询的最佳路径，但是它们为剪枝查询提供了有价值的参考值，实现了第一层次的计算共享；进一步的，核心子图机制在无需预处理的前提下，挖掘已有查询结果的最佳路径，实现对热路径的计算共享。和全局索引相比，核心子图更加的轻量级，因此可以通过增加热顶点数目做到更高的覆盖范围，提供更精确的剪枝界限值，加速剪枝查询的收敛速度。算法3展示了计算共享机制的伪代码。

Batch Execution of Similar Tasks

At any given moment, there are numerous random query tasks in the concurrent task pool, each with significantly different optimal query paths. We observed that some tasks exhibit low similarity, meaning their query tasks have minimal overlap, and in some cases, no overlap at all. Conversely, there are tasks with high similarity, where, for instance, the path of query A includes the path of query B, and for query B, its path completely overlaps with query A. For our data sharing mechanism, the higher the overlap rate of paths, the more efficient the data access, resulting in fewer redundant synchronous iterations. To address this, we propose a batch execution strategy that is aware of similar tasks, selecting batches of similar tasks from the task pool for execution. This approach further leverages data and computation similarity. Specifically, GraphCPP randomly selects a query task from the task pool, retrieves the starting and destination vertices of the task, and performs BFS algorithm to obtain the neighbor vertex sets SetS for the starting vertex and SetD for the destination vertex. Note that, considering that some central nodes may have a large number of neighbor nodes, we set an upper limit of 500 for neighbor nodes. Subsequently, it traverses the task pool, filtering out all queries with starting points in SetS and destination points in SetD, treating them as similar tasks to be processed concurrently. It's important to note that if the starting or destination vertex of a query belongs to a high-degree vertex, indexing can be directly used to accelerate the query process without employing regular query steps. Excluding high-degree vertices, the overhead of the k-hop SSSP itself is minimal, and the execution process can be concurrent with regular queries, making the execution cost negligible.

**Computation Sharing Mechanism**

GraphCPP achieves two levels of computation sharing through the global index mechanism and the core subgraph index mechanism. The inherent overhead of a global index is significant, so in practice, a small number of hot vertices are often chosen to build a global index (typically around 16). However, due to the characteristics of a power-law distribution, these hot vertices act as intermediary hub nodes for different queries. As a result, for the majority of point-to-point queries, a path through the source vertex, global index vertex, and destination vertex can be found. Although ensuring that this path is always the optimal path for a query is challenging, it provides valuable reference values for pruning queries, realizing the first level of computation sharing. Furthermore, the core subgraph mechanism, without the need for preprocessing, explores the optimal paths based on existing query results, achieving computation sharing for hot paths. Compared to the global index, the core subgraph is more lightweight, allowing for a higher coverage range by increasing the number of hot vertices. This provides more accurate pruning bounds, accelerating the convergence speed of pruning queries. Algorithm 3 illustrates the pseudocode for the computation sharing mechanism.

|  |
| --- |
| Algorithm 3: Shared Computation |
| 1: function IndexPreprocess(V, k, m): // V represents the set of graph vertices, with k and m indicating the counts of global index and core subgraph vertices, respectively.  2: globalVertices, coreSubgraphVertices = SortVerticesByDegree(V, k, m)  3: globalIndex = BuildGlobalIndex(k)  4: coreSubgraphIndex = InitializeCoreSubgraphIndex(m, globalIndex)  5: function SharedComputation(globalIndex, coreSubgraphIndex, query):  6: bound = FirstLevelSharedComputation(globalIndex, query) // Obtain the pruning bound for pruning queries through the global index.  7: while (activeVerticesCount != 0):  8: activeVertex = GetNextActiveVertex()  9: if activeVertex in coreSubgraph:  10: UpdateBounds(bound, SecondLevelSharedComputation(coreSubgraphIndex, query))  11: for neighbor of activeVertex:  12: UpdateBoundsByNeighbors(neighbor)  13: activeVerticesCount = UpdateActiveVertices()  14: function MaintainCoreSubgraph(bestPath):  15: hotPath = ExtractHotPath(bestPath)  16: hotPathValue = CalculateHotPathValue(hotPath)  17: AddToCoreSubgraph(hotPath, hotPathValue) |

实现计算共享的执行步骤如下：1，索引预处理（第1-4行），系统在对顶点的度数进行排序之后，选择度数最高的k+m个热顶点，将前k个顶点作为全局索引顶点（k值由用户确定），剩下的顶点作为核心子图顶点。全局索引的计算在预处理阶段完成，GraphCPP会执行SSSP算法计算k个高度顶点与图上的所有顶点的最佳路径（包含索引值和路径父节点），将结果存入以高度顶点id为索引的数组保存。核心子图省去了预计算过程，直接复用每次查询的计算结果，因此在预处理阶段只需要进行初始化。2，计算共享（第5-13行）：全局索引顶点充当了查询路径的枢纽节点，大部分查询都存在至少一条经过全局索引顶点的路径，虽然这条路径不一定是最佳路径，但是却为查询剪枝提供了一个可靠参考。因此每次执行点对点查询前，首先借助全局索引确定粗略的界限值，这是第一层次的计算共享。随后执行迭代查询算法，不断处理新的活跃顶点，直至所有顶点都收敛。对于每一个活跃顶点，我们判断它是否属于核心子图。初始时核心子图为空，所以不会参与共享，随着查询任务的执行，核心子图中逐渐新增了更多热路径。当活跃顶点属于核心子图成员时，就可以借助核心子图直接获取到对应起始顶点的热路径值，从而避免了对热路径的重复计算。此外核心子图可以让查询边界通过热路径直接从一个热顶点跳到另一个热顶点，加速了点对点查询的速度3，维护核心子图（第14-17行）：为了保证核心子图的轻量级，我们不对热路径进行预计算，而是通过从已有的最佳路径中发掘热路径的子集，来复用之前的计算结果。显然一条最佳路径上任意两个顶点之间的路径都是最佳路径，因此我们只需要以很小的开销，识别已有结果中的热顶点，并通过前缀和方法求得热顶点之间的结果即可。为了实现这一目的，我们需要在查询过程中保留遍历路径以及从源顶点出发到达每一个中间点的路径值，由于点对点查询本来需要计算这些信息，因此并不会造成额外的计算开销。通过上述步骤，我们用轻量级的核心子图索引，实现了高效地数据共享。The steps for implementing computation sharing are as follows:

1. \*\*Index Preprocessing (Lines 1-4):\*\* After sorting the degrees of vertices, the system selects the top k+m hot vertices with the highest degrees. The first k vertices serve as global index vertices (where k is user-defined), and the remaining vertices become core subgraph vertices. The computation of the global index is completed during preprocessing. GraphCPP executes the SSSP algorithm to calculate the optimal paths (including index values and parent nodes) for k high-degree vertices to all vertices in the graph. The results are stored in an array indexed by the high-degree vertices' IDs. The core subgraph omits the precomputation process, directly reusing the computation results of each query, requiring only initialization during preprocessing.

2. \*\*Computation Sharing (Lines 5-13):\*\* Global index vertices act as pivotal nodes for query paths. Most queries have at least one path passing through global index vertices. Although this path may not be the optimal path, it provides a reliable reference for query pruning. Therefore, before executing point-to-point queries, an approximate boundary is determined using the global index, representing the first level of computation sharing. Subsequently, an iterative query algorithm is executed, continuously processing new active vertices until all vertices converge. For each active vertex, it is determined whether it belongs to the core subgraph. Initially, the core subgraph is empty and does not participate in sharing. As query tasks execute, the core subgraph gradually accumulates more hot paths. When an active vertex belongs to the core subgraph, the hot path value for the corresponding starting vertex can be directly obtained through the core subgraph, avoiding redundant computation. Additionally, the core subgraph allows the query boundary to jump directly from one hot vertex to another through the hot path, accelerating the speed of point-to-point queries.

3. \*\*Maintain the Core Subgraph (Lines 14-17):\*\* To ensure the lightweight nature of the core subgraph, hot paths are not precomputed. Instead, a subset of hot paths is explored from existing optimal paths, reusing previous computation results. Clearly, any path between any two vertices on an optimal path is also an optimal path. Therefore, with minimal overhead, identifying hot vertices from existing results and calculating results between hot vertices using a prefix sum method is sufficient. To achieve this, traversal paths and path values from the source vertex to each intermediate point need to be retained during the query process. Since point-to-point queries inherently require calculating this information, the overhead is minimal. Through these steps, we achieve efficient data sharing using a lightweight core subgraph index.

**增量更新机制**

实际应用中，供查询任务遍历的底层图常常是动态变化（发生边增加eadd和边删除edelete）的，而变化的图结构数据可能会导致原有的全局索引和核心子图索引不再正确。因此，当动态图发生图更新时，我们除了需要更新图结构信息，还需要动态更新索引。图结构信息更新：GraphCPP通过邻接列表来存储每个顶点的出边邻居，因此我们只需要根据边增加（边删除）时的源顶点信息，修改对应的邻接表项即可；索引更新：我们采用一种增量更新的方式，依次更新全局索引和核心子图索引，尽可能减少了索引更新的冗余计算开销。

全局索引顶点数目k一般较小（通常设为16），但是记录的索引值非常多（索引值为k\*|V|），所以可以把索引信息存放在各个顶点上。每个顶点维护两个表格，table1记录了该顶点到k个全局顶点的最佳路径上的父节点，table2该顶点到k个全局顶点的最佳路径的索引值。我们根据边更新的类型，来对这两个表格进行增量更新。具体地，当发生边增加更新eadd时，我们首先获得更新边的源顶点src，目的顶点dst，以及两点之间的权重。接着我们依次检查每个全局索引顶点，如果Indexsrc+weight>indexdst，则更新table1中的parentdst为src，table2中的indexdst为Indexsrc+weight,否则无需更新该全局顶点索引；当发生边删除更新时，我们依次检查每一个全局索引顶点，判断parentdst是否等于src。如果是，表示我们删除了原本到达dst的最佳路径，此时需要重新计算indexdst。同时和其它的增量计算方法一样，对dst的更新会逐渐向外传递，所有经过dst的最佳路径的下游顶点都需要被更新。如果parentdst不等于src，则无需更新该全局索引顶点。

In practical applications, the underlying graph traversed by query tasks often undergoes dynamic changes, involving edge additions ($e\_{\text{add}}$) and edge deletions ($e\_{\text{delete}}$). Changes in the graph structure data can lead to errors in index values. Therefore, when dynamic updates occur in the dynamic graph, we not only need to update the graph structure information but also dynamically update the indexes. Graph structure information update: GraphCPP stores the out-neighbor of each vertex using an adjacency list. Therefore, we only need to modify the adjacency list of the corresponding out-neighbors based on the source vertex information when an edge is added (or deleted). Index update: We adopt an incremental updating approach, sequentially updating the global index and core subgraph index, minimizing redundant computation costs during index updates.

The number of global index vertices is relatively small ($k$ global index vertices), but it records a substantial number of index values ($k \times |V|$ global index values). Therefore, index information can be stored on each vertex. Each vertex maintains two tables: $table1$ records the parent nodes on the optimal paths to $k$ global vertices, and $table2$ records the index values of the optimal paths from this vertex to $k$ global vertices. Incremental updates are performed on these two tables based on the type of edge update. Specifically, when an edge addition update ($e\_{\text{add}}$) occurs, we first obtain the source vertex $src$, destination vertex $dst$, and the weight between the two points. We then sequentially check each global index vertex. If $Index\_{\text{src}} + \text{weight} > Index\_{\text{dst}}$, we update $parent\_{\text{dst}}$ in $table1$ to $src$ and $Index\_{\text{dst}}$ in $table2$ to $Index\_{\text{src}} + \text{weight}$. Otherwise, there is no need to update the index for that global vertex. In the case of an edge deletion update, we check each global index vertex and determine if $parent\_{\text{dst}}$ equals $src$. If true, it indicates that we have deleted the original optimal path to $dst$, and we need to recalculate $Index\_{\text{dst}}$. Similar to other incremental computation methods, updates to $dst$ will gradually propagate outward, requiring updates for all downstream vertices on the optimal paths that pass through $dst$. If $parent\_{\text{dst}}$ is not equal to $src$, there is no need to update that global index vertex.

The core subgraph index records a small number of indexes between high-degree vertices (up to $m \times m$ index values, where $m$ is significantly smaller than the graph data scale). Therefore, we use an independent two-dimensional array to store the core subgraph. Specifically, edge addition updates add to the existing graph structure, which may create new shortcuts, causing the original optimal paths to degenerate into non-optimal paths. For our pruning queries, non-optimal path indexes lead to early overestimation of boundary values. However, as the iteration progresses, point-to-point queries still traverse to more optimal paths, ultimately converging to the correct optimal paths. Extracting the latest hot path values from the converged paths completes the update to hot paths. For edge deletion updates, GraphCPP checks if both vertices of the deleted edge appear on some hot path. If yes, the original hot paths are interrupted, rendering all affected hot paths invalid. If only one vertex or no vertices appear on some hot path, the deleted edge does not affect hot paths, and there is no need for an update. Since the core subgraph index reuses the optimal path results from each query, no separate calculation is needed, resulting in overall low overhead.

The above mechanism implements incremental maintenance of graph structure data, global indexes, and core subgraph indexes. Considering that subtle graph updates do not significantly impact the overall computation results, we temporarily store subtle graph updates $\Delta G$ until its size exceeds a preset threshold or a certain time interval is reached. Only then do we execute batch graph update operations, further reducing update costs.

核心子图索引只记录少量热顶点之间的索引，最多只需维护m\*m个索引值（m的数量级远小于图数据规模），所以我们采用独立的二维数组来存储核心子图。具体地，边增加更新在原有的图结构上做加法，因此可能产生新的捷径，导致原有的最佳路径退化为非最佳路径。对于我们的剪枝查询来说，非最佳路径的索引会导致早期的界限值偏大，但是随着迭代的进行，点对点查询会依然会遍历到更优路径，最终会收敛到正确的最佳路径。此时从收敛路径中提取最新的热路径值，即可完成对热路径的更新；对于边删除更新，GraphCPP会判断删除边的两个顶点是否同时出现在某个热路径上。如果是，原有的热路径被中断，所有受影响的热路径都失效。如果只有一个顶点或者没有顶点出现在某个热路径上，则删除的边不会影响对热路径产生影响，此时无需更新。由核心子图索引复用了每次查询的最佳路径结果，无需专门计算，因此整体开销较小。

上述机制实现了对图结构数据、全局索引、核心子图索引的增量式维护。考虑到细微的图更新不会对整体计算结果产生大的影响，我们暂存细微的图更新∆G直到它的大小超出预设的阈值或者达到一定的时间间隔，才会批量执行图更新操作，从而进一步降低了更新开销。

实验评估

Experimental Setup

**硬件配置**。The experiments were conducted on an 8-node cluster, each machine equipped with 2 Intel Xeon E5-2680 v4 CPUs boasting 14 physical cores, 256 GB memory, and a 35MB LLC. All nodes were interconnected through an Infiniband network with a bandwidth of 300Gbps. The programs were compiled using gcc version 7.5.0, openMPI version 4.1.2, and with openMP enabled.(辛苦罗康来校正)

**图算法。**为了与之前的工作保持一致[xx]，我们选择了六种不同算法，它们被作为基准广泛应用于图聚类、图分类和图预测等领域。这些算法分为图算法和无向图算法。1）加权图算法：点对点最短路径（PPSP）、点对点最宽路径（PPWP）和点对点最窄路径（PPNP）分别用于找到两个顶点之间的之间的最短、最宽或最窄的路径。被广泛应用于于社交/金融网络、交通规划、监控洗钱行为和网络质量分析等领域；2）无权图图算法：广度优先搜索（BFS）、连接性和可达性和是未加权图上三个最常见的成对查询，分别用于确定两个顶点之间的最短路径、在无向图上，两个顶点是否连通、在有向图上两个顶点是否连通。它们被广泛应用于bi-connectivity、higher-order connectivity和graph clustering等高级算法。

**图数据集**。上述算法使用的图数据集如表x所示。其中LiveJournal和Twitter-2010属于社交网络图表。UK-2007-05和Gsh-2015-host为网页crawl图。它们都属于小直径和偏斜度分布的幂律图，贴合真实世界中的图分布情况。

EXPERIMENTAL EVALUATION

**Hardware Configuration**: The experiments were conducted on an 8-node cluster, with each machine equipped with 2 Intel Xeon E5-2680 v4 CPUs featuring 14 physical cores, 256 GB of memory, and a 35MB Last-Level Cache (LLC). All nodes were interconnected through an Infiniband network with a bandwidth of 300Gbps. The programs were compiled using gcc version 7.5.0, openMPI version 4.1.2, and with openMP enabled.

**Graph Algorithms:** To maintain consistency with prior work [xx], we selected six different algorithms widely applied as benchmarks in various graph-related fields such as graph clustering, graph classification, and graph prediction. These algorithms fall into the categories of weighted graph algorithms and unweighted graph algorithms. 1）Weighted Graph Algorithms: Point-to-Point Shortest Path (PPSP), Point-to-Point Widest Path (PPWP), and Point-to-Point Narrowest Path (PPNP) are employed to find the shortest, widest, or narrowest path between two vertices. These algorithms have broad applications in areas such as social/financial networks, traffic planning, money laundering monitoring, and network quality analysis. 2）Unweighted Graph Algorithms: Breadth-First Search (BFS), Connectivity, and Reachability are three common pairwise queries on unweighted graphs. They are used to determine the shortest path between two vertices, check whether two vertices are connected in an undirected graph, and verify connectivity between two vertices in a directed graph, respectively. These algorithms are widely applied in advanced algorithms like bi-connectivity, higher-order connectivity, and graph clustering.

**Graph Datasets**: The graph datasets used for the aforementioned algorithms are presented in Table x. LiveJournal and Twitter-2010 belong to social network graphs, while UK-2007-05 and Gsh-2015-host represent web-crawl graphs. These datasets exhibit power-law graphs with small diameters and skewed distributions, capturing real-world graph distribution scenarios. Our experiments are based on dynamic graphs, utilizing a snapshot mechanism where graph updates are performed on an unclosed snapshot, and graph queries are executed on a closed snapshot. Unclosed snapshots are periodically transformed into closed snapshots, replacing the original snapshot.

**系统比较**。We compared the query performance of GraphCPP with PnP, Tripoline, and SGraph. Since all of them are not open source, we re-implement their mechanism based on Gemini distributed graph processing framework. And bacause of none of these three systems directly support concurrent operations, we made some modifications to enable them to handle multiple jobs simultaneously. In the end, we compared the performance with two different modes of systems: PnP-S, PnP-C, Tripoline-S, Tripoline-C, SGraph-S, SGraph-C, . Among them, the systems with the "-S" suffix sequentially handles the jobs, while the systems with the "-C" suffix concurrently handle the jobs. In the systems with the "-C" suffix, the concurrent jobs are managed by the operating system.

To evaluate the performance, we sequentially or concurrently submit PPsP, PPWP, PPNP, BFS, Connectivity and Reachability in order until the specific number of jobs are generated. We set the parameters randomly for each job, even though these jobs may be the same graph algorithm. For example, the root vertices are randomly selected for the BFS jobs and the PPsP jobs. For concurrent submissions,任务的提交间隔也采取随机的方式. Tripoline、SGraph、GraphCPP采用了类似的全局索引机制，在实验中我们将全局索引的数目k设为16，将GraphCPP中的核心子图顶点设为128All benchmarks are executed 10 times, and the experimental results are reported as average value.**System Comparison**: We conducted a comparative analysis of GraphCPP's query performance with that of PnP, Tripoline, and SGraph. Since these systems are not open source, we re-implemented their mechanisms using the Gemini distributed graph processing framework. As none of these three systems inherently supports concurrent operations, we made modifications to enable them to handle multiple jobs simultaneously. We compared the performance of these systems in two modes: PnP-S, PnP-C, Tripoline-S, Tripoline-C, SGraph-S, and SGraph-C. In systems with the "-S" suffix, jobs are sequentially processed, while in those with the "-C" suffix, jobs are handled concurrently, managed by the operating system.

To evaluate performance, we submitted PPsP, PPWP, PPNP, BFS, Connectivity, and Reachability queries sequentially or concurrently until a specific number of jobs were generated. Parameters for each job were set randomly, even if they belonged to the same graph algorithm. For concurrent submissions, the intervals between job submissions were also randomized. Tripoline, SGraph, and GraphCPP employed a similar global index mechanism, with the global index set to 16 in our experiments. The core subgraph vertices in GraphCPP were set to 128. All benchmarks were executed 10 times, and the results are reported as average values.

整体性能对比

Figure 9 shows the total execution time of xx concurrent jobs with different schemes. 为了简洁起见，只显示最佳结果由于不同测试用例的执行时间差异较大，我们将归一化的执行时间表示为PnP的性能。It can be observed that for all graphs, GraphCPP achieves shorter execution times (thus higher throughput) than the other schemes. Compared to SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C, GraphCPP achieves an average throughput improvement of approximately xx, xx, xx, xx, xx, and xx times, respectively. The improvement of throughput is achieved by reducing the data access cost 和核心子图带来的高效剪枝in GraphCPP.

为了进一步地剖析性能，我们进一步地将总时间分成了数据访问时间和图处理时间。如图10所示，相比于其他系统，GraphCPP图数据访问占用的时间更少，并且随着图规模的增加该部分占用比例进一步减少。例如，对于Gsh-2015-host，GraphCPP的数据访问时间相比于其他系统分别减少了xx倍-xx倍。GraphCPP性能提升的原因有两个：1）不同并发作业所需图数据的相同部分在缓存中只需要加载和维护一个副本，减少了资源争用；2）图数据块根据关联任务数量的多少划分优先级，并定期地加载到LLC中被作业重用。优先处理优先级更高（关联任务数更多）能有效降低LLC miss率，减少不必要的内存数据传输。此外得益于两级计算共享机制，GraphCPP的计算时间也小于其他系统。Overall Performance Comparison

Figure 9 depicts the total execution time of xx concurrent jobs using different schemes. For conciseness, only the optimal results are displayed, and due to significant variations in the execution times of different test cases, we normalize the execution time relative to PnP's performance. It is evident that, for all graphs, GraphCPP achieves shorter execution times (and higher throughput) compared to the other schemes. In comparison to SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C, GraphCPP demonstrates an average throughput improvement of approximately xx, xx, xx, xx, xx, and xx times, respectively. This enhancement in throughput is accomplished by reducing data access costs and efficiently pruning the core subgraph in GraphCPP.

To provide a more in-depth analysis of performance, we further divide the total time into data access time and graph processing time. As shown in Figure 10, GraphCPP requires less time for graph data access compared to other systems, and this proportion decreases further as the graph size increases. For example, in the case of Gsh-2015-host, GraphCPP's data access time is reduced by xx times to xx times compared to other systems. Two key factors contribute to GraphCPP's efficiency: 1) identical portions of graph data required by different concurrent jobs only need to load and maintain a single copy in memory, reducing memory consumption; 2) graph data blocks are prioritized and periodically loaded into the LLC based on the number of associated tasks, promoting job reuse and effectively lowering LLC miss rates, minimizing unnecessary memory data transfers. Additionally, thanks to the two-level computing sharing mechanism, GraphCPP's computation time is also lower than that of other systems.

数据共享机制的效率

GraphCPP通过数据共享机制，减少了冗余的数据访问，为了定性地说明数据共享机制的效率，我们评估不同系统的LLC利用率，并将结果显示在图11中。从图中可以看出，GraphCPP的LLC miss率低于另外6个系统。在UK-2007-05中，GraphCPP的LLC miss率仅为xx，相比之下，SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C的LLC miss率分别为xx，xx，xx，xx，xx，and xx。这主要是因为在GraphCPP中，多个作业共享LLC中的一个图数据副本，能够更加充分地利用LLC，提高作业的数据局部性。

进一步地，我们还跟踪了并发任务交换到LLC中的数据总量。一般来讲，并发执行模式（-C）会比顺序执行模式（-S）交换更多的数据到LLC中，这是因为并发作业之间没有数据共享，不同作业之间强烈的缓存干扰会导致图数据在LLC中频繁的换入换出，产生更多的冗余内存数据传输。如图12所示，GraphCPP的数据交换量比SGraph-S，PnP-S，和Tripoline-S少得多（在UK-2007-05中分别是三者的xx，xx，and xx）。这是因为GraphCPP充分利用了并发任务之间的数据访问相似性。

计算共享机制效率

图x评估了每个系统图处理时间随时间变化的关系，由于每个系统图处理时间差异较大，我们将归一化的时间表示为PnP图处理时间。其中PnP纯粹依靠剪枝，没有用到索引，虽然减少了预处理时间，但是查询的计算时间也最长。Tripoline借助上界剪枝和全局索引减少了图处理时间。SGraph由于采用了上界+下界的剪枝策略，剪枝效果最好，需要计算的顶点最少，图处理时间也最少。这三个系统的图处理时间可能随着不同的查询特征而略有波动，但是整体上图处理时间稳定不变。而GraphCPP的图处理时间一开始和SGraph一致，因为它采用了和SGraph类似的全局索引机制，然而由于GraphCPP的核心子图索引机制会随着查询的进行不断完善，从而可以借助第二层级的优化不断加快图处理时间,GraphCPP的处理时间会 随时间不短缩短。**Efficiency of Data Sharing Mechanism:**

GraphCPP employs a data sharing mechanism to reduce redundant data access. To qualitatively illustrate the efficiency of this mechanism, we evaluate the LLC utilization of different systems, and the results are presented in Figure 11. Notably, GraphCPP demonstrates lower LLC miss rates compared to the other six systems. In the case of UK-2007-05, GraphCPP's LLC miss rate is only xx, while SGraph-S, SGraph-C, PnP-S, PnP-C, Tripoline-S, and Tripoline-C exhibit LLC miss rates of xx, xx, xx, xx, xx, and xx, respectively. This is primarily attributed to GraphCPP allowing multiple jobs to share a single graph data copy in the LLC, enabling more efficient utilization of the LLC and enhancing data locality for jobs.

Additionally, we track the total amount of data swapped into the LLC by these 16 jobs. Generally, the concurrent execution mode (-C) tends to swap more data into the LLC compared to the sequential execution mode (-S). This is because concurrent jobs lack data sharing, leading to frequent swapping of graph data in and out of the LLC and resulting in more redundant memory data transfers. As illustrated in Figure 12, GraphCPP swaps significantly less data compared to SGraph-S, PnP-S, and Tripoline-S (xx, xx, and xx, respectively, in UK-2007-05). This reduction is attributed to GraphCPP maximizing the similarity in data access among concurrent tasks.

Efficiency of the Computation Sharing Mechanism

Figure x evaluates the temporal relationship between the graph processing time of each system. Due to significant differences in the graph processing time of each system, we represent the normalized time as the PnP graph processing time. PnP relies purely on pruning without utilizing an index, reducing preprocessing time but resulting in the longest query computation time. Tripoline, leveraging upper-bound pruning and a global index, reduces graph processing time. SGraph, employing upper-bound and lower-bound pruning strategies, achieves the best pruning effect, requiring the fewest calculated vertices and minimizing graph processing time. While the graph processing time of these three systems may fluctuate slightly based on different query characteristics, overall, the graph processing time remains stable. In contrast, GraphCPP's graph processing time initially aligns with SGraph, as it adopts a global index mechanism similar to SGraph. However, due to the continuous improvement of GraphCPP's core subgraph index mechanism during queries, it can leverage optimizations at the second level to continuously accelerate graph processing time, resulting in a reduction over time; GraphCPP shortens its processing time.

**全局索引开销**。随着全局全局索引数目的增加，维护索引所需要的时间也不断。在我们的实验中，除了PnP由于没有采用全局索引而不会产生维护索引开销，Tripoline、SGraph、GraphCPP均采用了类似的全局索引机制。如图x所示，由于采用了相同的全局索引数目，GraphCPP构建索引的开销和其他系统持平。不过由于GraphCPP的吞吐量更高，相同时间可以处理更多查询任务，从而分摊了全局索引开销。

**核心子图索引开销**。我们也评估了核心子图索引对GraphCPP性能的影响。在全局索引数设置为16的情况下，GraphCPP-128，GraphCPP-256，和GraphCPP-without分别是核心子图索引选择128，256个顶点和不使用核心子图索引的版本。在表x中，我们展示了这三个版本核心子图占用内存空间的大小。可以看到GraphCPP-128和Graph-CPP-256相比于GraphCPP-without只增加了极少的内存空间占用（三者都维护16个全局索引）。这是因为相对于全局索引保存到所有顶点的距离，核心子图顶点只保存与其他核心子图顶点的距离，而这只需很少的额外空间。

此外，表x显示了GraphCPP-128，GraphCPP-256，和GraphCPP-without在16个作业上的总执行时间。我们可以观察到GraphCPP-256和GraphCPP-128总是优于GraphCPP-without，并且GraphCPP-256也比GraphCPP-128更快。在Friendster上，GraphCPP-256和GraphCPP-128的处理时间分别只有GraphCPP-without的xx和xx。这是因为增加的这些核心子图顶点能够有效地提高计算共享比例，减少冗余计算量。

**更新维护开销**。Tripoline、SGraph、GraphCPP三个系统的全局索引机制一脉相承，由于系统设置的全局索引数都是16，它们的全局索引更新开销都是一样的。同时GraphCPP还通过核心子图机制实现了第二层次的共享，因此它需要额外的更新开销。如xxx节所描述，当发生边增加更新时，无需专门更新对应的核心子图热路径。当发生边删除更新，只需要删除收影响的热路径。由于我们没有专门计算核心子图，而是从已有的计算结果中提取热路径。所以图更新只会影响核心子图机制的准确率，但是不会带来昂贵地更新维护开销。**Global Index Overhead:** As the number of global indices increases, the time required for maintaining the indices also grows. In our experiments, PnP did not incur maintenance overhead for global indices as it did not utilize them. However, Tripoline, SGraph, and GraphCPP employed similar global index mechanisms. As shown in Figure x, GraphCPP's index construction overhead is comparable to other systems due to the adoption of the same number of global indices. Nevertheless, GraphCPP achieves higher throughput, allowing it to handle more queries in the same time frame, thereby distributing the global index overhead.

**Core Subgraph Index Overhead**: We evaluated the impact of the core subgraph index on GraphCPP's performance. With the global index set to 16, GraphCPP-128, GraphCPP-256, and GraphCPP-without represent versions with core subgraph indices of 128, 256 vertices, and without using core subgraph indices, respectively. Table x illustrates the memory space occupied by these three versions of core subgraphs. It is noticeable that GraphCPP-128 and GraphCPP-256 incur only a minimal increase in memory usage compared to GraphCPP-without, all maintaining 16 global indices. This minimal increase is attributed to the fact that core subgraph vertices only store path value to other core subgraph vertices, requiring very little additional space compared to global indices storing path value to all vertices.

Moreover, Table x displays the total execution time of GraphCPP-128, GraphCPP-256, and GraphCPP-without across 16 jobs. Consistently, GraphCPP-256 and GraphCPP-128 outperform GraphCPP-without, with GraphCPP-256 being faster than GraphCPP-128. On Friendster, the processing times of GraphCPP-256 and GraphCPP-128 are only xx and xx times that of GraphCPP-without, respectively. This improvement is attributed to the additional core subgraph vertices enhancing pruning effectiveness and constraining upper and lower bounds.

**Update Maintenance Overhead**: Tripoline, SGraph, and GraphCPP share a common global index mechanism, with an equal global index number set to 16. GraphCPP, additionally implementing a second level of sharing through the core subgraph mechanism, incurs additional maintenance overhead. As described in Section xxx, during edge addition updates, there is no need to specifically update the corresponding core subgraph hot paths. In the case of edge deletion updates, only the affected hot paths need to be deleted. Since we do not perform a dedicated computation for the core subgraph but rather extract hot paths from existing results, graph updates only affect the accuracy of the core subgraph mechanism without incurring expensive update maintenance overhead.

可扩展性

图x（复用之前的图）显示了不同PPsP作业并发数目下GraphCPP和另外6个系统的性能比较。当并发作业数目增加时，GraphCPP可以得到更好的性能提升。当并发任务数分别为2、4、8和16时，GraphCPP相对于SGraph-S的加速分别为xx，xx，xx和xx。这是因为随着作业数目的增加，GraphCPP通过摊销节省了更多的数据访问和存储开销。请注意，当只有一个并发作业时，GraphCPP的细粒度调度操作不会发生，因此此时和其他方案相比执行时间没有太大差异。根据我们的测试，GraphCPP上块调度成本只占总执行时间的xx-xx%。此外，由于对LLC等资源的争夺，并发版本（-C）的性能要比GraphCPP甚至顺序版本（-S）差得多。因此，简单地修改现有的图处理系统来支持并发任务可能是一个糟糕的选择。

然后我们评估了GraphCPP的横向扩展性。为了实现这一目标，我们首先在单个节点上增加CPU核数来评估GraphCPP在16个PPsP作业上的执行时间。从图x中可以看出，GraphCPP的性能在任何情况下都要好于其他方案，特别是在核数较多的情况下。这是因为GraphCPP的图数据是由并发任务共享的，而其他方案的数据访问成本更高。其次，我们还评估了1、2、4、8节点上不同方案的性能。如图18所示，GraphCPP在8个节点上的性能是单机的xx-xx倍，具有良好的扩展性。此外，GraphCPP的扩展性也要好于SGraph，PnP和Tripoline，这是因为数据共享和核心子图索引使得通信成本更低。因此我们相信GraphCPP可以高效地支持实际中的点对点查询应用。

Scalability

Figure 16 illustrates the performance comparison between GraphCPP and six other systems under varying numbers of concurrent PPsP jobs. GraphCPP exhibits superior performance improvement as the number of concurrent jobs increases. For 2, 4, 8, and 16 concurrent jobs, GraphCPP achieves acceleration ratios of xx, xx, xx, and xx, respectively, in comparison to SGraph-S. This improvement stems from GraphCPP's amortization, which saves more data access and storage costs with an increasing number of jobs. It's important to note that with only one concurrent job, GraphCPP's fine-grained scheduling operations do not occur, resulting in minimal differences in execution time compared to other schemes. According to our tests, the block scheduling cost of GraphCPP constitutes only xx-xx% of the total execution time. Furthermore, due to resource contention, the performance of the concurrent version (-C) is considerably worse than both GraphCPP and the sequential version (-S). Therefore, a simple modification of existing graph processing systems to support concurrent tasks might not be an optimal choice.

Next, we assess the horizontal scalability of GraphCPP. To achieve this goal, we initially increase the number of CPU cores on a single node to evaluate GraphCPP's execution time with 16 PPsP jobs. As depicted in Figure 17, GraphCPP consistently outperforms other schemes, especially when the number of cores is higher. This is attributed to the shared graph data among concurrent tasks in GraphCPP, reducing data access costs compared to other schemes. Additionally, we evaluate the performance of different schemes on 1, 2, 4, and 8 nodes. As shown in Figure 18, GraphCPP's performance on 8 nodes is xx-xx times that of a single machine, indicating robust scalability. Moreover, GraphCPP's scalability surpasses that of SGraph, PnP, and Tripoline due to lower communication costs facilitated by data sharing and core subgraph indexing. Consequently, we believe that GraphCPP can efficiently support practical point-to-point query applications.

相关工作

**图计算系统**。随着图计算逐渐成为研究热点，有越来越多不同类型的图计算系统被提出。1）单机内存图计算系统：Ligra[x]通过push和pull计算模式的切换，加快图计算特别是图遍历算法的收敛。KickStarter[x]和GraphBolt[x]实现了对流式图的增量处理。DiGraph[x]提出一种GPU上的高效的迭代有向图处理系统。LCCG[x]、DepGraph[x]和TDGraph[x]利用拓扑感知执行方法，减少了图计算的冗余计算和数据访问成本。2）单机核外图计算系统：GraphChi[x]和X-Stream[x]通过顺序访问存储实现高效的核外图形处理。FlashGraph[x]采用半外部内存图形引擎，实现高IOPS和并行性。GridGraph[x]提高了局域性并减少了I/O操作。 DGraph[x]通过更快的状态传播加速图处理。GraphM[x]采用数据共享的思想，优化并发系统的吞吐量。3）分布式图计算系统。Pregel[x]提出BSP计算模型，在性能，可扩展性和容错方面满足了具有billions规模的边的图的处理。PowerGraph[x]提出了基于“顶点切割”（vertex-cut）的图划分思想，有效地减少了通信量以及由度数较高顶点导致的负载不均衡。PowerLyra[x]混合计算模型，针对高度顶点和低度顶点采用不同计算策略。Gemini[x]提出了一个双模式计算引擎，在确保性能的前提下实现了可扩展性。CGraph[x]使用以数据为中心的LTP（load-trigger-pushing）模型，利用并发任务之间的时间/空间相似性，降低了分布式系统的开销。上述工作都针对通用图算法而设计，然而由于点对点查询算法只关注特定顶点对之间的图关系，它和通用图算法有着显著不同。最直观地一点，点对点查询算法可以通过剪枝策略极大地提高执行效率，然而通用的图计算系统却并不支持。RELATED WORK

**Graph Processing Systems**: With the increasing prominence of graph computing as a research focus, a plethora of graph computing systems of diverse types have emerged. 1)Single-node in-memory graph computing systems: Ligra [x] accelerates graph computation, especially graph traversal algorithms, by switching between push and pull computation modes. KickStarter [x] and GraphBolt [x] implement incremental processing for streaming graphs. DiGraph [x] proposes an efficient iterative directed graph processing system on GPUs. LCCG [x], DepGraph [x], and TDGraph [x] utilize topology-aware execution methods, reducing redundant computation and data access costs; 2) Single-node out-of-core graph computing systems: GraphChi [x] and X-Stream [x] achieve efficient out-of-core graph processing through sequential storage access. FlashGraph [x] employs a semi-external memory graph engine, achieving high IOPS and parallelism. GridGraph [x] improves locality and reduces I/O operations. DGraph [x] accelerates graph processing through faster state propagation. GraphM [x] adopts a data-sharing approach, optimizing throughput for concurrent systems; 3) Distributed graph computing systems: Pregel [x] introduces the Bulk Synchronous Parallel (BSP) computation model, addressing the processing needs of graphs with billions of edges in terms of performance, scalability, and fault tolerance. PowerGraph [x] proposes a "vertex-cut" graph partitioning approach, effectively reducing communication volume and addressing load imbalance caused by high-degree vertices. PowerLyra [x] employs a hybrid computing model, using different computation strategies for high-degree and low-degree vertices. Gemini [x] introduces a dual-mode computation engine, ensuring scalability while maintaining performance. CGraph [x] uses a data-centric Load-Trigger-Pushing (LTP) model, leveraging temporal and spatial similarities between concurrent tasks to reduce distributed system overhead; While the aforementioned works are designed for general graph algorithms, point-to-point query algorithms, focusing on specific vertex pairs' graph relationships, exhibit significant differences. Point-to-point query algorithms, through pruning strategies, can greatly enhance execution efficiency. However, general graph computing systems do not inherently support such algorithms.

**点对点查询**。现有工作对点对点查询做出了许多研究，硬件方面，𝐻𝑢𝑏2 [x]提出了一种以高度顶点为中心的专用加速器，通过限制高度顶点的搜索范围以及对搜索过程剪枝来加速PPSP过程。软件方面，Quegel[x]走索引路线，通过在图加载时构建分布式图的静态索引，来提高查询的相应速度。PnP[x]走剪枝路线，通过一个通用的剪枝策略，减少点对点查询的过程中的冗余访问和冗余计算。Tripoline[x]同时结合索引和剪枝，利用三角不等式原理实现了无需先验知识的查询的增量评估。SGraph[x]进一步优化剪枝策略，利用基于上界和下界的剪枝策略，实现了在大型图上以亚秒级的延迟查询不断变化的数据。上述工作专注于通过剪枝和索引等机制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图计算进行了研究。例如对于单机内存图处理系统。Congra[x]通过离线分析查询任务的内存带宽消耗和原子操作特征，在不因资源争用而阻塞查询的前提下动态调度任务，以获得更好的系统吞吐量和资源效率。Krill[x]提出了一种SAP模型，将图结构、算法和属性解耦。利用属性缓冲区轻松地写入和管理属性数据，利用图核融合技术将所有作业作为一个整体，来减少内存访问。ForkGraph[x]提出了一个高效的缓冲区执行模型，来实现并发任务的的数据共享，并且采用基于让步的调度策略加速了整体执行速度。对于单机核外图计算系统。GraphM[x]提出利用并发任务的时间和空间局部性实现并发图计算的数据共享。对于分布式图计算系统。Seraph[x]提出将图结构数据与任务特定数据解耦，以允许并发任务共享公共图结构数据。MultiLyra[x]和BEAD[x]通过图共享和边界共享，分摊集群中计算节点之间的通信成本，从而支持高效的批处理查询评估。CGraph[x]是GraphM[x]的分布式版本。上述方案专注于对并发任务的数据访问进行共享优化，忽略了并发任务之间的计算优化，且没有针对点对点查询进行专门优化。

**Point-to-Point Queries**: Previous studies have delved into extensive research on point-to-point queries. Notably, 𝐻𝑢𝑏2 [x] introduced a specialized accelerator with a hub-centric approach, emphasizing the challenges posed by vertices with a high degree of connections, or hubs, in expanding the search space for shortest path calculations. To address this, it proposed the hub-Network concept, limiting the search scope of hub nodes and accelerating the PPSP process through online pruning. However, due to the specialized nature of 𝐻𝑢𝑏2's dedicated accelerator, its applicability is constrained. 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 integrated both indexing and pruning, employing the triangle inequality principle to achieve incremental evaluation of point-to-point queries without prior knowledge. SGraph optimized pruning strategies further, using upper and lower bounds based on the triangle inequality principle, achieving sub-second query response times on large graphs. Despite these advancements, these systems predominantly concentrate on enhancing the speed of individual point-to-point queries, overlooking the substantial load posed by large-scale concurrent queries.

**Concurrent Graph Computing**: Various graph computing systems have explored the realm of concurrent computing. GraphM identified "data access similarity" among concurrent graph computing tasks and proposed a data-centric scheduling strategy to facilitate data sharing between multiple tasks, enhancing the throughput of concurrent graph computing. However, GraphM is a single-machine out-of-core graph computing system using the BSP computing model, limited to static graphs. Building upon this, CGraph[x] extended the application scenarios to distributed dynamic graph computing systems, optimizing communication mechanisms and load balancing strategies for distributed scenarios. Despite these advancements, it remains an out-of-core system, not well-suited for high-load concurrent query scenarios, even with the distribution of disk access costs across different subgraphs through scheduling strategies. ForkGraph efficiently conducts concurrent graph processing in memory, employing a concession-based scheduling strategy, handling only a portion of the data in each iteration to accelerate overall execution speed. Nevertheless, it is a single-machine in-memory system, not specifically optimized for point-to-point queries, making it unsuitable for executing concurrent point-to-point query tasks on massive datasets.

结论

现有的点对点查询系统聚焦于优化单次查询的速度，忽略了对并发查询吞吐量的优化。本文发现并发点对点查询任务存很强的数据访问相似性和计算相似性，并提出了一个数据驱动的并发点对点查询系统GraphCPP。它通过数据驱动的缓存执行机制，实现并发查询之间的重叠数据的访问共享。同时通过一个双层级计算共享机制，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

The existing point-to-point query systems primarily focus on optimizing the speed of individual queries, neglecting the optimization of throughput for concurrent queries. This paper identifies strong data access similarity and computation similarity among concurrent point-to-point query tasks. It proposes a data-driven concurrent point-to-point query system, GraphCPP. By employing a data-driven caching execution mechanism, GraphCPP achieves overlapping data access sharing among concurrent queries. Simultaneously, through a two-level computation sharing mechanism, it better realizes computation sharing among multiple tasks. Experimental results indicate that GraphCPP's performance surpasses that of the state-of-the-art graph query systems, including SGraph[x], Tripoline[x], and Pnp[x], by a factor of XXX.

ACKNOWLEDGMENTS

参考文献

1. Chen H, Zhang M, Yang K, et al. Achieving Sub-second Pairwise Query over Evolving Graphs[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 2023: 1-15.
2. Jiang X, Xu C, Yin X, et al. Tripoline: generalized incremental graph processing via graph triangle inequality[C]//Proceedings of the Sixteenth European Conference on Computer Systems. 2021: 17-32.
3. Xu C, Vora K, Gupta R. Pnp: Pruning and prediction for point-to-point iterative graph analytics[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 587-600.
4. <https://www.alibabagroup.com/document-1595215205757878272>
5. [www.google.com/maps](http://www.google.com/maps)
6. [www.facebook.com](http://www.facebook.com)
7. [www.alipay.com](http://www.alipay.com)
8. <http://www.cagis.org.cn/>
9. [www.baike.baidu.com](http://www.baike.baidu.com)
10. [www.ditu.amap.com](http://www.ditu.amap.com)
11. [www.http://map.qq.com/](http://www.http://map.qq.com/)
12. [www.petalmaps.com](http://www.petalmaps.com)
13. Gemini: A Computation-Centric Distributed Graph Processing System
14. Avery C. Giraph: Large-scale graph processing infrastructure on hadoop[J]. Proceedings of the Hadoop Summit. Santa Clara, 2011, 11(3): 5-9.
15. Low Y, Gonzalez J E, Kyrola A, et al. Graphlab: A new framework for parallel machine learning[J]. arXiv preprint arXiv:1408.2041, 2014.
16. Gonzalez J E, Low Y, Gu H, et al. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs[C]//10th USENIX symposium on operating systems design and implementation (OSDI 12). 2012: 17-30.
17. Gonzalez J E, Xin R S, Dave A, et al. {GraphX}: Graph Processing in a Distributed Dataflow Framework[C]//11th USENIX symposium on operating systems design and implementation (OSDI 14). 2014: 599-613.
18. Xie C, Chen R, Guan H, et al. Sync or async: Time to fuse for distributed graph-parallel computation[J]. ACM SIGPLAN Notices, 2015, 50(8): 194-204.
19. Chen R, Shi J, Chen Y, et al. Powerlyra: Differentiated graph computation and partitioning on skewed graphs[J]. ACM Transactions on Parallel Computing (TOPC), 2019, 5(3): 1-39.

废弃材料

废弃摘要内容：

在面对高并发的点对点查询需求时，由于冗余的数据访问，处理效率很低。我们观察到并发查询任务之间存在着数据访问相似性，这启发我们提出了一种以数据为中心的并发点对点查询方法。具体地，我们将图查询过程中的数据分为“图结构数据”和“任务特定数据”，前者记录了图的拓扑信息，后者记录了查询任务所要访问的图结构数据分块，不同查询独立访问任务所需的数据分块，这些分块可能重叠，但在传统的查询方案中。因此，我们采用了一种数据驱动的调度方法：在执行并发点对点查询任务时，内存/LLC中只保留一份图结构数据。多任务之间以细粒度的图数据分块为单位共享数据。一次访问，多个任务处理，以此分摊数据访问的开销，提高并发图查询的吞吐量。为了展示GraphCPP的效率，

核心子图查询机制

如PnP使用基于下界的剪枝方法来减少查询过程中的冗余访问；Tripoline通过维护中心顶点到其它顶点的日常索引，实现无需先验知识的快速查询；SGraph利用三角不等式原理，提出了基于“上界+下界”的剪枝方法，进一步减少点对点查询过程中的冗余访问；

素材库：

CGP 作业固有的不规则访问导致由于局部性较差而导致底层内存子系统利用率不足。最终导致整个系统的吞吐量较低。首先，CGP作业由于其不同的遍历特性，对相同图结构数据表现出不规则的图遍历，并且这些作业同时访问同一图的不同部分。来自多个作业的这种不规则且不协调的内存访问会导致严重的缓存抖动。其次，CGP 作业对内存子系统造成激烈的资源争用。当在现有的多核处理器上运行多个作业时，这些作业会将与同一顶点关联的状态提取到不同的缓存行中。由于图的稀疏性，每个缓存行中只需要几个数据元素（甚至一个），因为图处理因展示对小数据元素（例如，每个顶点状态 4 或 8 字节）的固有随机访问而臭名昭著。整个图 [14,25,39]。

这会导致激烈的资源争用以及缓存和内存带宽的利用不足。

单作业加速器对于解决 CGP 作业之间不协调的图形数据访问效率低下

LCCG通过新的硬件机制增强了众核处理器：图遍历正则化和预取。前一个组件规范了 CGP 作业的图遍历，从根本上解决这些作业的不规则数据访问的挑战。与遍历正则化相结合，预取组件进一步隐藏了 CGP 作业的内存延迟，并有效地支持这些作业的合并访问。具体来说，顶点状态沿着图拓扑中固有的依赖关系传播。仅当其状态由其活动邻居更新时，非活动顶点才需要由作业处理。基于这一见解，提出了一种有效的拓扑感知执行方法，并得到 LCCG 的有效支持。它根据图拓扑动态探索所有 CGP 作业的以活动顶点为根的公共遍历路径，然后预取这些探索路径上的图数据，以驱动相应的作业一起同步处理这些数据。

CGP 作业会发出更多的冗余数据访问，并且由于不同作业在不同时间将更多冗余数据存储到缓存中，也会导致更严重的缓存干扰。它最终会导致系统吞吐量低下，因为数据访问成本通常占迭代图算法总执行时间 的主要部分。

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

**数据集**

|  |  |  |  |
| --- | --- | --- | --- |
| Datasets | Vertices | Edges | Data sizes |
| LiveJournal[1] | 4.8M | 69M | 526MB |
| Twitter-2010[2] | 41.7M | 1.5B | 10.9GB |
| Gsh-2015-host[3] | 68.7M | 1.8B | 13.4GB |
| UK-2007-05[4] | 106M | 3.74B | 27.9GB |

[1] 2019.SNAP.http://snap.stanford.edu/data/index.html.

[2] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In Proceedings of the 19th International Conference on World Wide Web (Raleigh, North Carolina, USA) (WWW ’10). Association for Computing Machinery, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[3] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2018. BUbiNG: Massive Crawling for the Masses. ACM Trans. Web 12, 2, Article 12 (jun 2018), 26 pages. https://doi.org/10.1145/3160017

[4] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. SIGIR Forum 42, 2 (December 2008), 33–38. https://doi.org/10.1145/1480506.1480511

算法

为了与之前的工作保持一致[xx]，我们选择了六种不同算法，它们被广泛应用于图聚类、图分类和图预测等领域。这些算法分为图算法和无向图算法。

* 加权图算法：
  + 点对点最短路径（PPSP）、点对点最宽路径（PPWP）和点对点最窄路径（PPNP）分别用于找到两个顶点之间的之间的最短、最宽或最窄的路径。被广泛应用于于社交/金融网络、交通规划、监控洗钱行为和网络质量分析等领域
* 无权图图算法
  + 广度优先搜索（BFS）、连接性和可达性和是未加权图上三个最常见的成对查询，分别用于确定两个顶点之间的最短路径、在无向图上，两个顶点是否连通、在有向图上两个顶点是否连通。它们被广泛应用于bi-connectivity、higher-order connectivity和graph clustering等高级算法。
* 点对点最短路径（PPSP）是单源最短路径（SSSP）的“成对查询”版本。它用于发掘任意两对顶点之间的最短路径，被广泛应用于社交/金融网络等领域。
* 点对点最宽路径（PPWP）和点对点最窄路径（PPNP）用于找到两个顶点之间的之间的最宽或最窄的路径，被广泛应用于交通规划，监控洗钱行为和网络质量分析等领域
* 例如，单源最短路径（SSSP）是典型的单源查询应用程序，它计算从源到所有其他目标顶点的最短路径。相比之下，SSSP 的动态“成对查询”版本称为点对点最短路径（PPSP），其中用户只对一对两个任意顶点之间的最短路径感兴趣。 PPSP 不仅是动态道路图上的导航系统 [57] 中最重要的问题，而且也是社交/金融网络中许多高级图分析应用的基本构建块。
* 例如，推荐系统需要频繁检查从购物日志中提取的大型网络中两个任意用户之间的最短路径，风险检测系统使用 PPSP 来测量新到达的交易与某些风险用户之间的距离[22， 28、35]。

我们评估了竞争系统在 BC、LL 和 NCP 三种应用程序上的性能。为了与之前的工作保持一致[1,18,47]，我们对三个应用程序进行如下配置。

七种不同应用的多功能集，即最短路径 (PPSP)、广度优先搜索 (BFS)、可达性、连通性、最宽路径 (PPWP)、最窄路径 (PPNP) ）和维特比算法（Viterbi）。所有这些应用程序都是重要的成对图查询，1）不仅它们本身被广泛使用； 2）而且这些基本内核的组合可以形成许多重要的高级图分析应用，例如聚类、分类和预测。

**PPSP：**

Henan Wang, Guoliang Li, Huiqi Hu, Shuo Chen, Bingwen Shen, Hao Wu, WenSyan Li, and Kian-Lee Tan. 2014. R3: A Real-Time Route Recommendation System.

Proc. VLDB Endow. 7, 13 (aug 2014), 1549–1552. <https://doi.org/10.14778/2733004.2733027>

Q. Guo, F. Zhuang, C. Qin, H. Zhu, X. Xie, H. Xiong, and Q. He. 5555. A Survey on Knowledge Graph-Based Recommender Systems. IEEE Transactions on Knowledge and Data Engineering 01 (oct 5555), 1–1. https://doi.org/10.1109/TKDE.2020.3028705

Kevin Joseph and Hui Jiang. 2019. Content Based News Recommendation via Shortest Entity Distance over Knowledge Graphs. In Companion Proceedings of The 2019 World Wide Web Conference (San Francisco, USA) (WWW ’19). Association for Computing Machinery, New York, NY, USA, 690–699.https://doi.org/10.1145/3308560.3317703

**PPWP：**

Maurice Pollack. 1960. Letter to the Editor—The Maximum Capacity Through a Network. Operations Research 8, 5 (1960), 733–736. <https://doi.org/10.1287/opre.8.5.733>

**PPNP：**

Oded Berman and Gabriel Y Handler. 1987. Optimal minimax path of a single service unit on a network to nonservice destinations. Transportation Science 21, 2 (1987), 115–122. <https://doi.org/10.1287/trsc.21.2.115>

**Reachability**

Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. SIAM J. Comput. 32, 5 (2003), 1338–1355.https://doi.org/10.1137/S0097539702403098

Liam Roditty and Uri Zwick. 2008. Improved Dynamic Reachability Algorithms for Directed Graphs. SIAM J. Comput. 37, 5 (2008), 1455–1471. https://doi.org/10.1137/060650271

Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD ’14). Association for Computing Machinery, New York, NY, USA, 1323–1334. <https://doi.org/10.1145/2588555.2612181>

**Connectivity**

Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. arXiv:2008.03909 Retrieved August 14, 2020 from <https://arxiv.org/abs/2008.03909>

**Viterbi**

Satu Elisa Schaeffer. 2007. Graph clustering. Computer Science Review 1, 1 (2007), 27–64. <https://doi.org/10.1016/j.cosrev.2007.05.001>

**BFS**

Satu Elisa Schaeffer. 2007. Graph clustering. Computer Science Review 1, 1 (2007), 27–64. <https://doi.org/10.1016/j.cosrev.2007.05.001>