图计算系统存储控制与优化综述

谈安东[1)]

[1)](华中科技大学 计算机科学与技术学院, 武汉市 430070)

摘 要 对于大型图计算系统而言，如何存储其计算时所需要的边权节点信息，以及如何优化存储结构，都是值得深入思考的问题。在过去的研究中许多不同的方法被提出，大方向如扩大存储容量，改变存储方式，运用核外存储，使用分布式存储等，而这些又能分为很多更细小更有针对性的优化方向。本篇综述围绕几篇基于图计算系统存储优化原理的论文展开，讲述了不同图计算系统对应不同的优化方案是如何被提出、如何实施的。GRASP 在图分析的最后一级缓存（LLC）上进行抓取的、领域专门化的缓存管理；HEP 通过将图的边缘集分为两个子集来灵活地适应其内存开销，可以划分部分适合内存的图，同时产生较高的划分质量；ForkGraph 将图划分为每个大小的 LLC（最后一级缓存）容量的分区，并且分叉处理模式查询在分区的基础上进行缓冲和执行，以提高缓存重用；HytGraph 采用混合传输管理（HyTM）来最大化性能，提高 GPU 的处理速度；NVMGraph 将最频繁访问的分区中的随机访问的数据块从 NVM 迁移到 DRAM 做到减轻混合内存系统中分区之间的内存访问不平衡。

关键词 图计算系统；存储优化；图划分；并行处理；缓存重用

中图法分类号 TP338.8

A survey of storage control and optimization in graph computing systems

Andong Tan[1)]

[1)](Department of Computing, Huazhong University of Science and Technology, Wuhan 430070, China)

Abstract For a large graph computing system, how to store the edge weight node information required for its calculation and how to optimize the storage structure are all issues worthy of in-depth consideration. In the past, many different methods have been proposed, such as expanding storage capacity, changing storage mode, using off core storage, using distributed storage, etc., which can be divided into many smaller and more targeted optimization directions. This review focuses on several papers based on the storage optimization principle of graph computing systems, and describes how different graph computing systems corresponding to different optimization schemes are proposed and implemented. GRASP performs domain specific cache management on the last level cache (LLC) of graph analysis; HEP can flexibly adapt its memory overhead by dividing the edge set of the graph into two subsets, and can partition some graphs that are suitable for memory, while producing high partition quality; ForkGraph divides the graph into partitions of each size's LLC (last level cache) capacity, and the forked processing mode queries are buffered and executed on the basis of partitions to improve cache reuse; HytGraph uses Hybrid Transmission Management (HyTM) to maximize performance and improve the processing speed of the GPU; NVMGraph migrates randomly accessed data blocks in the most frequently accessed partitions from NVM to DRAM to reduce memory access imbalance between partitions in a mixed memory system.

Key words Graph Computing System; Storage Optimization; Graph Partition; Parallel processing; Cache reuse

本课题得到华中科技大学数据中心课程施展老师，童薇老师, 胡燏翀老师的帮助。谈安东，男，2000 年生，研究生，CCF 学生会员,主要研究领域为高性能图计算系统.E-mail: adtan@hust.edu.cn.

1 引言

事实上，在大规模图数据上进行分析并不是一个最近才出现的需求。超大规模集成电路的设计、运输路线的规划、电力网络的仿真模拟等等，都需要将数据抽象成图的表示并用到各种面向图的分析算法。

在“图计算系统”的概念出现前，完成这类任务通常需要针对每个场景编写相应的专用程序。借助已有的程序库可以减少不少工作量，例如大名鼎鼎的 Boost 中包含的专门面向图计算的 BGL（Boost Graph Library） 和 PBGL （ Parallel Boost Graph Library）。BGL 提供了用于表示图的数据结构以及一些常用的图分析算法；PBGL 则扩展了 BGL，在此之上基于 MPI 提供了并行/分布式计算的能力。CGMgraph 与 PBGL 类似，基于 MPI 提供了一系列图分析算法的并行/分布式实现。然而，早期的这些面向图计算的程序库缺乏对用户友好的编程模型，需要介入和管理的细节较多，上手难度较大。于是乎了，基于 GAS（Gather-Apply-Scatter）编程模型的图计算系统应运而生。

但在图计算系统开发之余，如何设计系统内部存储方式、如何使得图算法运行速度有保证且保持存储负载均匀，都是值得深思的课题。图计算具有数据依赖性强、访存－计算比高的特点，提高图计算访存效率是改善系统性能的关键. 尤其是随着图数据规模的扩大, 高效的存储管理优化对图计算性能的提高显得尤为重要。

现阶段有多种方法实现优化内存管理，诸如使用集成电路（ASIC、FPGA 等）、使用 BRAM、ReRAM 等片外存储器作为存储介质、设计使用新的内存架构（例如使用暂存器）等方法。直接替换或使用更高端硬件的优化相对更加直接、更加有针对性，而在设计逻辑层面基于存储层次结构的优化则更需要去深入图计算系统本身研究，且可以运用或重用在更多的图处理引擎中，更具有普适性，也是本文着重讨论的领域。

本篇综述旨在通过几篇文章提出的存储思路与方案来讨论存储管理优化在图计算系统中的可能性。

2 原理

2.1 GRASP

图分析为金融、网络和商业物流等不同领域的应用提供了动力。在图分析领域中使用的图的一个常见性质是顶点连通性的幂律分布，其中少量的顶点负责图中所有连接的很大比例。这些富连接、热的顶点天生就表现出高重用性。然而，这项工作发现，由于图分析的高度不规则的访问模式，最先进的硬件缓存管理方案难以利用它们的重用。为此，建议在图分析的最后一级缓存上进行抓取的、领域专门化的缓存管理。GRASP 增强现有的缓存策略，通过保护热顶点免受缓存抖动，最大限度地提高热顶点的重用，同时保持足够的灵活性，以根据需要捕获其他顶点的重用。GRASP 通过利用轻量级的软件支持来精确定位热顶点，从而使硬件成本可以忽略不计，从而避免了对最先进的缓存管理方案所采用的存储密集型预测机制的需要。

为了克服现有硬件缓存管理方案的局限性，我们提出了 GRASP 末级缓存（LLC）管理。据我们所知，这是第一个为图分析领域引入领域专门的缓存管理的工作。GRASP 增强现有的缓存插入和命中升级策略，为包含热顶点的缓存块提供优先处理，以保护它们免受抖动。为了满足不规则的访问模式，GRASP 被设计为可以灵活地缓存其他显示重用的块，基于观察到的访问模式提高缓存效率。

GRASP 依赖于现有的倾斜感知重排序软件技术，通过在一个连续的内存区域中分离热顶点来诱导空间局部性。虽然这些技术在重新排序成本和保留图结构的能力方面提供了不同的权衡，但它们都是通过将热顶点与冷顶点隔离来工作的。

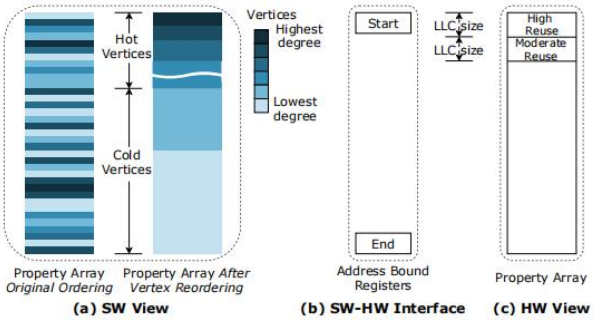


图 2.1 GRASP 的软件与硬件及交互层

图中说明了 GRASP 软件与硬件的设计思路。(a)软件应用顶点重新排序，它在数组的开头分离热顶点。(b)软-硬件接口为每个属性数组公开一个

ABR 对，以配置为数组的边界。由几个可配置寄存器组成，该寄存器软件在应用程序初始化期间用属性数组的边界填充。(c)根据 LLC 的大小来识别显示不同重用的区域。

热顶点划分逻辑：GRASP 基于可用的 LLC 容量，对仅包含热顶点子集（仅包含最热顶点）的缓存块进行优先级排序。属性数组开始处的 LLC 块大小内存区域被标记为 High Reuse Region；在高重用区域之后立即开始的另一个 LLC 大小的内存区域被标记为 Moderate Reuse Region。通过将地址与每个属性数组的 Reuse Region 的边界进行比较，根据地址所属区域来判断其 reuse 程度，将分类结果（高重用、中等重用、低重用或默认）编码为 2 位重用提示，并将其与每个缓存请求一起转发给 LLC，可指导后面专门的插入和命中提升策略。

专用缓存策略：GRASP 的组件实现了专门的缓存策略，以保护与 HighReuse LLC 访问相关联的缓存块免受抖动。一种简单的方法是将高重用的缓存块固定在 LLC 中。但是，固定将牺牲利用其他缓存块（例如，中度重用缓存块）可能暴露的时间重用的任何机会。为了克服这一挑战，GRASP 采用了一种灵活的方法，将现有的缓存替换策略增加为 LLC 命中的专门插入策略和 LLC 命中的升级策略。专门策略为高重用块提供了优先处理，同时保持在其他缓存块中利用时间重用的灵活性。标记为高重用的访问，包括属于高重用区域的最热顶点集，被插入到 MRU 位置的缓存中，以保护它们不受抖动。与 HighReuse LLC 访问相关联的缓存块会立即被提升到 MRU 位置，以保护它们免受冲击。一方面，这些块被进一步重用的可能性是非常有限的，这意味着它们不应该被直接地提升到 MRU 位置。另一方面，通过经历至少一次命中，这些区块显示了不能完全忽略的时间局部性。把握为这些街区的中间地带，逐步促进它们向 MRU 位置。

Table II
POLICY COLUMNS SHOW HOW GRASP UPDATES PER-BLOCK 3-BIT RRPV COUNTER OF RRIP (BASE SCHEME) FOR A GIVEN REUSE HINT. HIGHER RRPV VALUE INDICATES HIGHER EVICTION PRIORITY.

Reuse Hint	Insertion Policy	Hit Policy
High-Reuse	RRPV = 0	RRPV = 0
Moderate-Reuse	RRPV = 6	if RRPV > 0:
Low-Reuse	RRPV = 7	RRPV - -
Default	RRPV = 6 or 7	RRPV = 0

图 2.2 对各种热度顶点的 RRPV 值

上表显示对于给定的重用提示，GRASP 怎样更新 RRIP(基本方案)per-block3bit 的 RRPV 值。RRPV 值越高，逐出优先级越高。

Table V
PROPERTIES OF THE GRAPH DATASETS. TOP FIVE DATASETS ARE USED IN THE MAIN EVALUATION WHEREAS THE BOTTOM TWO DATASETS ARE USED AS ADVERSARIAL DATASETS.

Dataset	Vertex Count	Edge Count	Avg. Degree
LiveJournal (*lj*) [38]	5M	68M	14
PLD (*pl*) [4]	43M	623M	15
Twitter (*tw*) [54]	62M	1,468M	24
Kron (*kr*) [32]	67M	1,323M	20
SD1-ARC (*sd*) [4]	95M	1,937M	20
Friendster (*fr*) [23]	64M	2,147M	33
Uniform (*uni*) [62]	50M	1,000M	20

图 2.3 GRASP 使用数据集

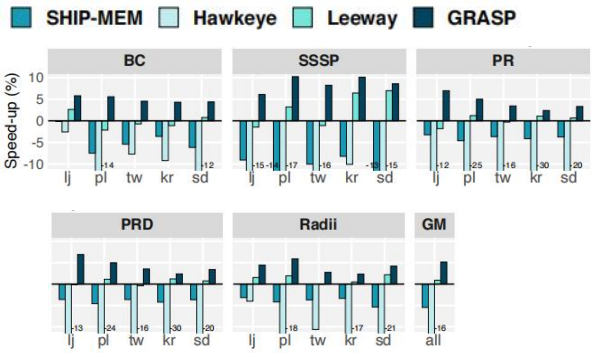


图 2.4 GRASP 实验结果

对于实验结果，不难发现在各种数据集中，GRASP 对于计算加速优于 Leeway（一种基于历史的缓存管理方案，基于称为实时距离的度量应用死块预测），远好于 Hit Predictor （SHiP，最先进的插入策略，它基于 RRIP）和 Hawkeye（最先进的缓存管理方案）。

2.2 HEP

HEP（Hybrid Edge Partitioner）可以划分部分适合内存的图，同时产生较高的划分质量。HEP 可以通过将图的边缘集分为两个子集来灵活地适应其内存开销。一个子集由 NE++的一种新的高效内存算法进行划分，而另一个子集由流式方法进行划分。我们对大型真实世界图的评估表明，在许多情况下，HEP 同时优于内存分区和流分区。

现有图划分主要分为内存划分（In-Memory Partitioning）与流划分（Streaming Partitioning）。前者划分结果质量较高，但所需存储空间大，成本昂贵；后者划分质量较低，且一次只访问图的一条边，效率低下。而 HEP 的思路就是将图划分为两块子图，一块使用内存划分，另一块使用流划分。与此同时，只有使用内存划分的子图被加载进内存中。大多数图都遵循一个规律：低度数节点多。HEP 致力于减少低度数节点的复制工作。

NE 算法：从随机种子 v 开始，将其放入核心

集合 C，将 v 个邻居移动到辅助集合 S，将 C 和 S 中顶点之间的边添加到当前分区，再展开将外部度数最低的顶点从 S 移动到 C，并将其外部邻居移动到 S。将 C 和 S 中顶点之间的边添加到当前分区继续展开，直到分区有足够的边缘。

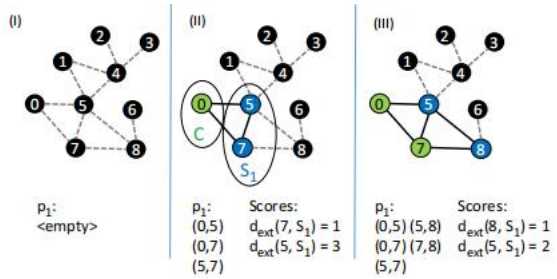


图 2.5 NE 算法分区

而 NE++通过以下优化扩展 NE（邻居扩展）：图修剪，无效化惰性边缘，优化种子顶点搜索，适配分区容量限制，构建最后一个分区等。

图修剪：加载图形时，NE++修剪高阶顶点，跳过高阶顶点的邻接列表，并将高阶顶点之间的边写入外部文件，如图所示。

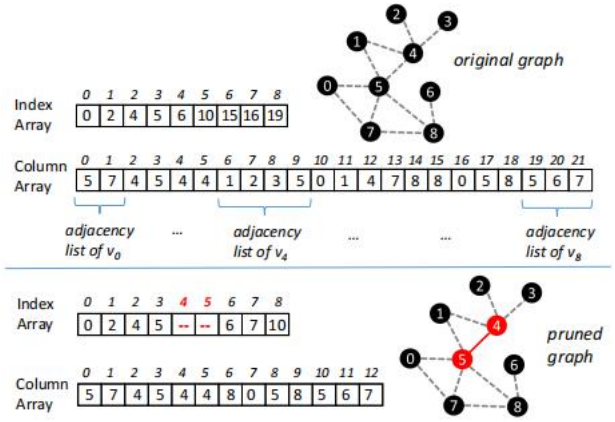


图 2.6 图修剪示例

惰性边缘无效化：NE 算法会记录所有指定边，而这不是必须的。NE++选择不再访问集合 C 中的顶点，因为 C 中顶点之间的边不是关键边；它将会再次访问 S 中的顶点，通过访问顶点访问到临界边（与 S 中顶点相关的边）。

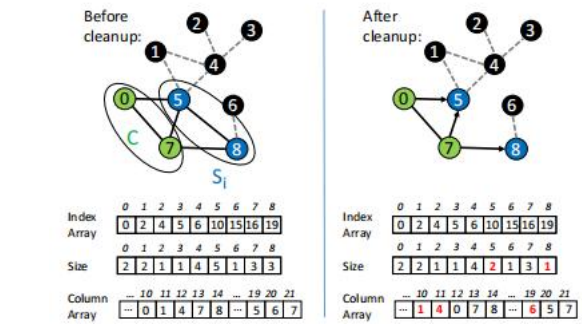


图 2.7 图修剪前后 CSR 表示

种子顶点搜索：当启动分区扩展时/当一个断开连接的零部件已指定所有边时/当只存在高阶顶点时，需要对其初始化。初始化开始时，需要找到种子顶点。在 NE 算法中，种子顶点是随机选择的，很难找到合适的顶点，需要许多随机“跳跃”直到找到合适的顶点；而 NE++算法按顶点 ID 顺序扫描，在整个种子顶点搜索中，每个顶点最多访问一次。

更多优化：构建最后一个分区时，只需将所有剩余的内存边缘分配给最后一个分区。在算法中，若当前的边有高度数邻居，则将这些内存边缘分配，构成最末分区。

Algorithm 3 Building the last partition in NE++.

```
1:  procedure AssignRemainingInMemoryEdges
2:      for each v ∈ V do
3:          if v ∉ C then
4:              for each u ∈ N_out(v) do
5:                  p_i ← p_i ∪ (u, v)
6:              if v has high-degree neighbors then
7:                  for each u ∈ N_in(v) do
8:                      if u ∈ V_h then
9:                          p_i ← p_i ∪ (v, u)
10:         if |p_i| ≥ |E|/k then
11:             i ← i + 1
```

图 2.8 NE++算法末级分区构建

与 HEP 混合分区的一个独特特点是它能够通过设置τ来灵活地减少内存开销。设置的τ越低，越多的顶点被认为是高度的，因此，列数组——这是主导的数据结构——被更积极地修剪。因此，τ可以根据计算节点的内存容量来设置。为此，我们可以执行一个预计算步骤，并为不同的τ值建立各自的低度顶点的邻接列表大小的累积和；这是一个简单的并行过程。然后，选择保持内存绑定的τ的最大值（例如，计算节点的内存容量）。一旦确定了τ，就使用 HEP 进行图分区。

在实验中，使用以下指标来衡量 HEP 分区是否有效：重复因子，分区运行时长，内存开销。

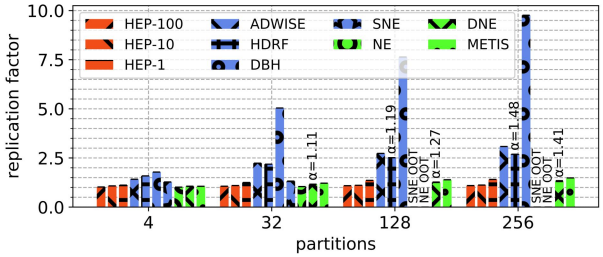


图 2.9 重复因子比较

在重复因子方面，HEP 表现比流式分区更好，与内存分区相比也有一定竞争力。设置τ越大时，重复因子对应也更小，表现更优秀。

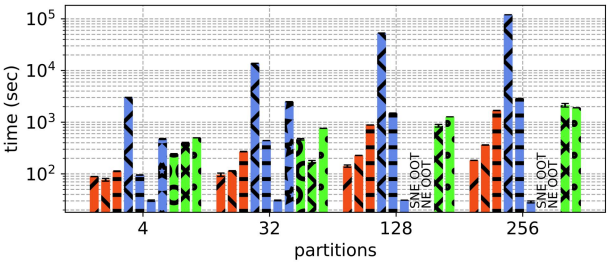


图 2.10 运行时长比较

在运行时长方面，与其他分区器相比，只有基于哈希的分区（DBH）更快，HEP 优于其他分区器。运行时间随减少而增加τ。

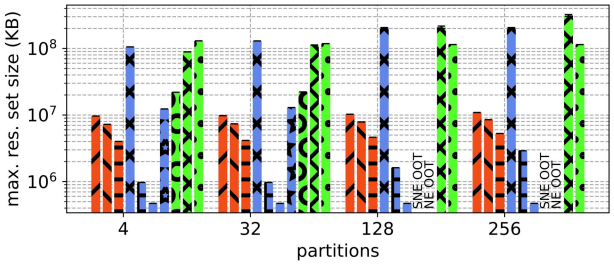


图 2.11 运行开销比较

在运行开销方面，HEP 内存开销比其他内存分区器普遍更低。其开销也与可调谐τ有关，τ较低，则内存开销较低。

2.3 ForkGraph

为了提高处理 FPP（分叉处理模式查询）的缓存效率，我们提出了一种缓存高效的系统，即格式图。格式图的核心设计是基于一种新的缓冲执行模型。我们将图 G 划分为 LLC 大小的分区，并将每个分区与存储 FPP 查询的操作到该分区的缓冲区关联起来。缓冲执行模型通过对来自不同查询的批处理操作来利用 FPP 查询之间的时间位置，并对每个分区分批执行它们。由于每个图分区都可以适应 LLC，因此批处理中操作的随机内存访问在 LLC 中自然受到限制，缓存错过率较低。

ForkGraph 包含了高效的分区内和分区间处理策略。对于分区内处理，工作效率变得至关重要，因为大多数操作都是在缓存驻留的图分区中处理的。因此，顺序实现来同时执行多个操作。通过分配一个线程来处理每个缓冲操作来利用查询间的并行性。采用顺序实现，因为它们通常比并行算法更高效。此外，ForkGraph 还整合了属于同一查询的缓冲区中的操作，因此，属于同一查询的操作可以以无原子的方式进行处理。此外，这种以查询为中心的操作整合大大减少了冗余操作。

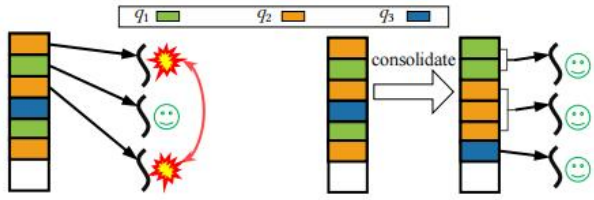


图 2.12 ForkGraph 分区内处理过程示例

分区内处理：假设有三个线程。在不整合的情况下，线程会在缓冲区中获取操作并同时处理它们。当两个线程处理来自同一个查询 q2 的操作时，需要应用原子操作来解决潜在的读写冲突。通过合并缓冲的操作来执行操作并将属于同一个查询的那些操作分配给一个线程，从而以无原子的方式执行操作处理。

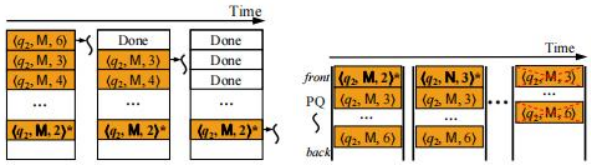
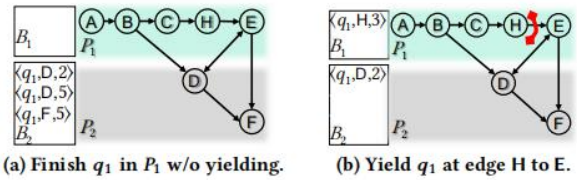


图 2.13 SSSP 修建冗余示例

上图给出了一个处理 SSSP 查询的合并操作的示例，可以利用用户定义的优先级函子来减少冗余操作的数量。在没有优先级化线程的情况下，逐个获取处理操作。具有最显著值的操作可以位于缓冲区的任何位置。<q2,M,2>包含最优值，并在许多其他操作之后进行排队。<q2,M,2>之前的操作都为冗余。相比之下，通过利用 Dijkstra 算法的优先级函子来解决 SSSP，ForkGraph 处理同一查询的最有利的操作。此处，首先对<q2,M,2>的处理可以有效地修剪冗余的部分。

分区间处理：采用启发式轮转调度来决定何时终止当前分区并开始下一分区的处理，采用优先级调度基于优先级大小来选择下一分区。



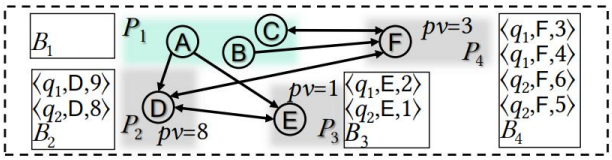
(a) Finish q_1 in P_1 w/o yielding. (b) Yield q_1 at edge H to E.

Figure 7: Comparison of the execution and number of operations with and without yielding in P_1. Shorest path query q_1 starts at vertex A in P_1. All edges are with unit lengths.

图 2.14 ForkGraph 分区间处理过程

上图显示了在有和没有产生影响的处理过程中执行和操作数量的比较。该图反映了完成加工 P1

和切换到 P2 的状态。在图 a 中，当 ForkGraph 完成 P1 中的所有操作并发送三个操作到 P2 时，它会切换到 P2。然后，它被安排去处理 P2。我们可以期待格式图必须重新访问和更新顶点 E. P1 再次因为最短路径将通过顶点。在图 b 中，ForkGraph 对顶点 H 启发式调度后，在 P1 中存储操作 H，并向 P2 发送一个操作。当 ForkGraph 在 P2 中完成执行时，它将操作<q1,E,3>中的最短路径更新发送给 E。与不产生的执行相比，冗余操作<q1,D,5>和<q1,F,5>被修剪。表格将恢复处理 P1 缓冲区中剩余的操作<q1,H,3>，以及从 P2 发送的操作。



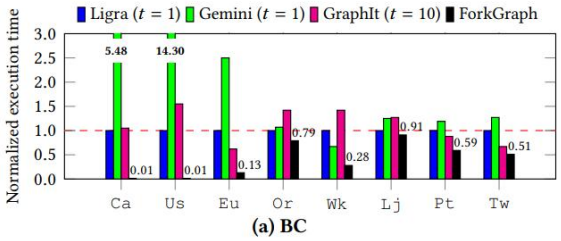
Scheduling	Execution order	#Operations processed
Random	$P_1,P_2,P_4,P_2,P_3,P_2,P_4$	11
Max #operations	P_1,P_4,P_2,P_3,P_2,P_4	9
FIFO	P_1,P_3,P_4,P_2,P_4	7
Priority-based	P_1,P_3,P_2,P_4	6

图 2.15 不同调度方案比较

上图说明了不同调度方式下的操作数。随机调度选择任意的分区，这可能比其他方法采取更多的步骤，导致收敛速度慢；Max 操作调度选择操作数最多的分区，最大限度地重用缓存内容。但由于涉及更多的冗余工作，它比其他方法要慢；基于 fifo 的调度基于生成的操作顺序访问分区；与这些方法相比，利用 Dijkstra 算法中的优先级函子的优先级调度可以最大限度地利用最短路径，减少冗余工作。从结果上看，优先级调度的操作数是最少的。

选择分区时，应该尽可能选择有利于全局信息快速收敛的分区。每个分区定义一个优先级，许多经典的图算法提供了大量的优先级函子（priority functor）设计，可以直接重用——例如图中的 Dijkstra 作为优先级函子等。

实验在 3 个图处理系统（Ligra,Gemini,GraphIT）上进行，图处理应用选择 NCP,BC 和 LL。结果如下：



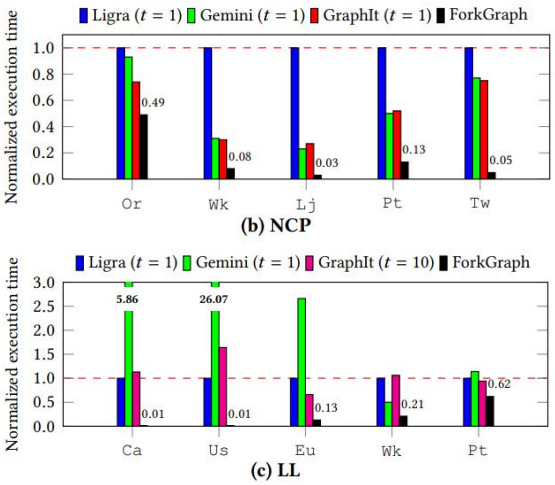


图 2.16 ForkGraph运行时长比较

比较运行时长，可以看到，ForkGraph 性能相较于其他三个图处理系统，分别提升了 32 倍,307 倍以及 38 倍，提升性能效果相当好。

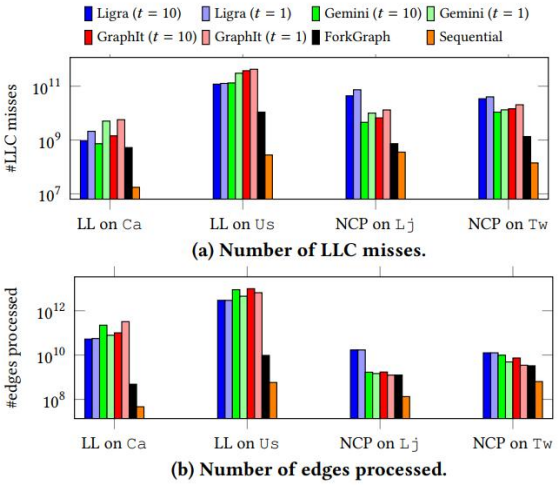


图 2.17 ForkGraph末级缓存与处理边数

ForkGraph 的末级缓存缺失比远小于其他图处理系统，且处理的边数也远远少于其他三种图处理系统，I/O 效果较好。

3 扩展

在研究各种缓存控制方法与调度方法之余，也可将这些与硬件加速结合，例如 HyTGraph（利用 GPU 进行加速与优先级调度结合），NVMGraph（利用非易失性存储器与工作窃取方法结合）等等。在此对两者作简短描述：

3.1 HyTGraph

HyTGraph 是一个 gpu 加速的图处理框架，采用混合传输管理（HyTM）来最大化性能。HyTGraph

将图组织成 CSR 结构，其邻居索引数组驻留在 GPU 全局内存中，并且与边相关的数组（邻居数组和边权数组）在主机端进行逻辑分区。根据现有的框架，HyTGraph 将与边缘相关的数据划分为 N 个边缘平衡分区{P0，P1…，PN−1}与基于块的分区，其中每个 Pi 都是分区 i 的一组连续编号的顶点。在迭代计算过程中，对包含活动边缘的分区使用其最经济有效的引擎进行 GPU 计算的调度。HyTGraph 提供了两个功能来实现高效的 HyTM：

1.成本感知任务生成

具有成本感知能力的任务生成模块中，HyTGraph 会计算不同方法的数据传输成本，并为每个分区选择成本效益最高的引擎。此外，HyTGraph 还提供了一个任务组合器，将子图（待计划的）合并到更大的任务中，以在任务调度阶段实现更低的调度开销。

2.异步任务调度

HyTGraph 公司引入了异步技术来提高任务调度效率。HyTGraph 不是简单地多次重新计算加载的子图，而是采用贡献驱动的优先级调度来对那些对收敛贡献更大的分区进行优先排序。该方法基于以下观察结果：那些具有较大程度的顶点往往成为计算路径中的枢纽。为了提高资源利用率，HyTGraph 使用多个 CUDA 流来重叠计算内核、数据传输和基于 cpu 的活动子图压缩。

HyTGraph 的结构如下图所示。

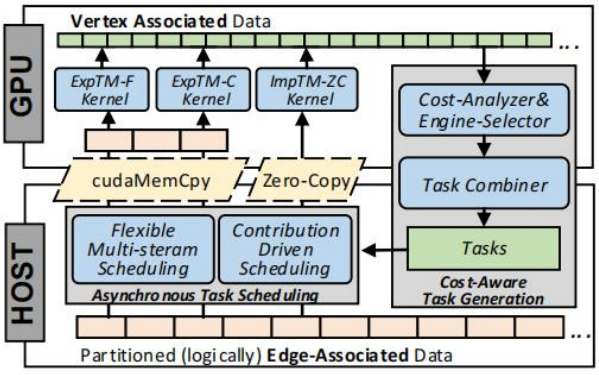


图 3.1 HyTGraph结构

使用 GPU 协作，以加速图形处理。

HyTGraph 这种高效的 gpu 加速图处理框架通过自适应地切换传输管理方法，包括显式传输管理和隐式传输管理，最大限度地提高了主机-GPU 的带宽，并且是实现最短的总体执行时间所必需的。

3.2 NVMGraph

NVMGraph 通过数据迁移解决了内存负载不平衡的问题。它包括两个步骤：算法导向的负载分析，和性能模型导向的数据迁移。在每次迭代中，我们在下一个迭代中分析每个分区的计算负载，以找到访问最频繁的分区。然后，基于一个性能模型进行数据迁移决策。

NVMGraph 会在运行时监视工作负载和迁移数据。我们在应用程序层实现了工作负载监控机制。它主要包括：1)监控每个分区的计算负载，2)找到一个计算负载最大的分区。负载信息被用作性能模型的输入，以决定数据迁移。图分区方案只能保证一个线程所需的随机访问的数据被合并到单个分区中，而其他的数据仍然可以跨不同的分区进行访问。监视分区的计算负载需要通常跨分区访问的状态数组，这限制了重叠工作负载监视和应用程序执行的机会。为此，NVMGraph 设计了周期性的和阈值触发的负载监控机制，以降低工作负载监控的成本。

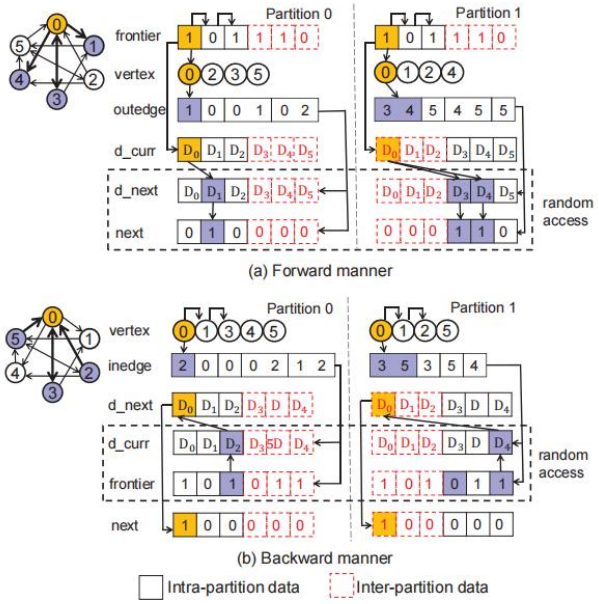


图 3.2 NVMGraph图划分与前后向遍历过程

图为 NVMGraph 的内存数据结构与执行流程。顶点值数据和状态数据根据其顶点（每个分区的顶点 id 集）分布到不同的分区。例如，值数据 d[0]到 d[0][2]，状态数据边界[0]到边界[2]分布在分区 0 中，而 d[0][3]到 d[0][5]，边界[3]到边界[5]分布在分区 1 中。图分区方案将一个线程随机访问的所有数据合并在一个分区中，以消除数据依赖性。它将跨分区随机数据访问转换为分区内数据访问，从而方便了数据迁移操作。例如，以正向的方式，线程

0 连续处理顶点 0、2、3 和 5，并且只更新分区 0 中的值数据（d 下一个[0]到 d 下一个[2]）和状态数据（下一个[0]到下一个[2]）。以反向的方式，线程 0 逐个处理顶点 0、1、3、4 和 5，并且只访问分区 0 中的值数据（d curr[0]到 d curr[2]）和状态数据（边界[0]到边界[2]）。类似地，分区 1 由线程 1 以同样的方式处理。

具有工作负载监控和数据迁移机制的图处理方案 NVMGraph 可用来解决内存负载不平衡问题。NVMGraph 首先通过依赖感知的图分区将工作线程所需的随机访问数据合并到单个分区中，然后通过分离感知器的数据迁移来平衡分区的内存负载。实验结果表明，与现有的 Ligra 相比，NVMGraph 可以提高高达 40.3%的应用性能。

4 总结

本文综述了几种图计算系统的存储控制优化方案，并结合了相对简单或复杂的图计算系统案例进行分析。不同系统中往往采用不同方案进行存储控制，例如采用将图划分为 LLC 大小块、使用启发式调度、优先级调度、使用工作监控/窃取、采用不同子图不同加载方式等各种优化方法。此外，这些方法也可以和硬件的使用结合，比如 GPU 加速或 NVM 存储优化等等。在将来的研究中，如何将图计算系统的硬件与软件优化、设施与策略优化更好地相结合，会一直是一个可用于突破图计算系统运行性能瓶颈的课题。

致　谢 感谢施展老师、童薇老师与胡燏翀老师为我们讲授数据中心技术这门课程，也感谢课程提供的学术会议模拟机会，让我得以巩固相关知识点并锻炼临场组织语言发挥的能力。

参 考 文 献

[1] Faldu, P.; Diamond, J.; Grot, B. Domain-Specialized Cache Management for Graph Analytics, IEEE International Symposium on High Performance Computer Architecture (HPCA).

[2] Mayer, R.; Jacobsen, H.-A. Hybrid Edge Partitioner: Partitioning Large Power-Law Graphs under Memory Constraints, SIGMOD/PODS '21: Proceedings of the 2021 International Conference on Management of Data, p 1289-302, 9 June 2021

[3] Shengliang Lu; Sun, S.; Paul, J.; Li, Y.; Bingsheng He .Cache-Efficient Fork-Processing Patterns on Large Graphs.: SIGMOD/PODS '21: Proceedings of the 2021 International Conference on Management of Data, p 1208-21, 9 June 2021

[4] Wang, Qiange; Ai, Xin; Zhang, Yanfeng; Chen, Jing; Yu, Ge. HyTGraph: GPU-Accelerated Graph Processing with Hybrid Transfer Management: arXiv, August 31, 2022

[5] Wei Liu (Huazhong University of Science and Technology, Service Computing Technology and System Laboratory, China); Haikun Liu; Xiaofei Liao; Hai Jin; Yu Zhang. Straggler-Aware Parallel Graph Processing in Hybrid Memory Systems:.2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), p 217-26, 2021