软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):1018−1042 [doi: 10.13328/j.cnki.jos.006449]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

RGraph: 基于 RDMA 的高效分布式图数据处理系统*

崔鹏杰[1], 袁 野[2], 李岑浩[1], 张 灿[1], 王国仁[2]

[1](东北大学 计算机科学与工程学院, 辽宁 沈阳 110819)

[2](北京理工大学 计算机学院, 北京 100081)

通信作者: 袁野, E-mail: yuanye@mail.neu.edu.cn

摘 要: 图是描述实体间关系的重要数据结构, 被广泛地应用于信息科学、物理学、生物学、环境生态学等重要的科学领域. 现如今, 随着图数据规模的不断增大, 利用分布式系统来处理大图数据已经成为主流, 出现了形如 Pregel、GraphX、PowerGraph 和 Gemini 等经典的分布式大图数据处理系统. 然而, 与当前先进的基于单机的图处理系统相比, 这些经典的分布式图处理系统在处理真实的图数据时并没有充足或稳定的性能优势. 分析了几个有代表性的分布式图处理系统, 总结并归纳出了影响其性能的主要挑战. 通过对这些挑战的深入研究, 提出了 RGraph——一个基于 RDMA 的高效分布式大图数据处理系统. RGraph 旨在通过充分利用 RDMA 的优势来提升图处理系统多个方面的性能. 在图划分方面, RGraph 采用基于块的划分方式避免破坏原始图数据的局部性, 从而保证顶点的高效访问. 在负载方面, RGraph 提出了基于 RDMA 单边 READ 的任务迁移机制和线程间细粒度的任务抢夺方式来分别保证计算节点间以及计算节点内线程间的动态负载均衡, 确保集群中的所有计算资源能够被充分利用. 在通信方面, RGraph 通过对 IB verbs 的有效封装, 实现了符合图计算语义的多线程 RDMA 通信模型. 相比于传统的 MPI, RGraph 的通信机制可以减少计算节点间 2.1 倍以上的通信延迟. 最后, 利用 5 个真实大图数据集和 1 个合成数据集, 在拥有 8 个计算节点的高性能集群上测试了 RGraph. 实验结果表明, RGraph 具有明显的性能优势. 相比于 Powergraph, RGraph 具有 10.1–16.8 倍的加速比, 与当前最先进的分布式图处理系统相比, RGraph 的加速比仍能达到 2.89–5.12 倍. 同时, RGraph 在极度偏斜的幂律图上也能保证稳定的性能优势.

关键词: 分布式; 图处理系统; 高性能; RDMA; 动态负载均衡; RDMA 通信模型

中图法分类号: TP311

中文引用格式: 崔鹏杰, 袁野, 李岑浩, 张灿, 王国仁. RGraph: 基于 RDMA 的高效分布式图数据处理系统. 软件学报, 2022, 33(3): 1018–1042. http://www.jos.org.cn/1000-9825/6449.htm

英文引用格式: Cui PJ, Yuan Y, Li CH, Zhang C, Wang GR. RGraph: Effective Distributed Graph Data Processing System Based on RDMA. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 1018–1042 (in Chinese). http://www.jos.org.cn/1000-9825/6449.htm

RGraph: Effective Distributed Graph Data Processing System Based on RDMA

CUI Peng-Jie[1], YUAN Ye[2], LI Cen-Hao[1], ZHANG Can[1], WANG Guo-Ren[2]

[1](School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China)

[2](School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

Abstract: Graph is a significant data structure which describes the relationship between entries, and it is widely used in information science, physics, biology, environmental ecology and other scientific fields. Nowadays, with the growing magnitude of graph data, processing large-scale graph data using distributed system has become the popular, many specialized distributed systems, including Pregel, GraphX, PowerGraph, and Gemini have been proposed. However, compared with the current state-of-the-art shared-memory graph

* 基金项目: CCF-华为数据库创新研究计划(DBIR2019007B)

本文由“数据库系统新型技术”专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.

收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

processing systems, these specialized distributed graph processing systems do not deliver satisfactory or stable performance advantages in processing real-world graph datasets. Several representative distributed graph processing systems are analyzed, and the major challenges that affect their performance are summarized. This study proposes RGraph, an effective distributed graph processing system based on RDMA. The key idea of RGraph is improving performance on top of making full use of the advantages of RDMA. For graph partition, RGraph adopts chunk-based partition to avoid destroying the native locality of the real-world graph, so as to ensure the locality-preserving vertex accesses. For workload, RGraph proposes a task migration mechanism based on RDMA one-side READ and a fine-grained task preemption method among threads to ensure the dynamic load balance for inter-node and intra-node, so that all computing resources can be fully utilized. For communication, RGraph effectively encapsulates IB verbs and implements a concurrent RDMA communication stack satisfied graph computing semantics. Compared with traditional MPI, RGraph's communication stack can reduce the latency up to 2.1 times for servers' communication. Finally, five real-world large-scale graph datasets and one synthetic dataset are used to evaluation RGraph on an HPC cluster with eight servers, and the experiment shows that RGraph has obvious performance advantages. Compared with Powergraph, RGraph has 10.1−16.8 times performance improvement. And compared with the existing state-of-the-art CPU- based distributed graph processing system, RGraph still has 2.89−5.12 times performance improvement. Meanwhile, RGraph can still guarantee stable performance advantage on extremely skewed power-law graph.

Key words: distributed; graph processing system; high performance; RDMA; dynamic load balance; RDMA communication model

随着社交网络等新型应用的兴起和云计算等新技术的快速发展, 人类获取数据的规模正以前所未有的速度爆炸式增长, 分析并挖掘海量数据间的关联性, 成为当前各行业的关注点[1]. 在技术探索中, 大图数据凭借出色的表示能力, 成为大数据时代下无处不在的数据结构. 图是表示对象与对象之间关系的方法, 一个图由若干顶点和连接它们的边组成, 是一种基于事务关联关系的模型表达. 基于图存储的图处理算法, 如广度优先遍历 BFS (beadth first search)、单元最短路径 SSSP (single source shortest path)、网页排名 PR (PageRank)、弱连通分量 WCC (weakly connected component)被广泛地应用于现实生活中的各个领域, 包括运输行业的滴滴打车[2]、电子商用中的欺诈检测[3]、社交网络中的用户测评[4]、知识图谱中的查询应答[5]、智慧城市中的城市监控[6]等.

近年来, 随着各领域的图数据不断积累, 图数据规模已经达到了 TB 级, 并向 PB、EB 级不断增大. 例如, 用于记录网页数据的 ClueWeb[7]公开数据集在 2012 年就达到了 10 亿顶点、425 亿条边, 文件大小达到了 1.2 TB 的规模. 因此, 基于单机的图处理系统无论在计算层面还是存储层面, 已经不能满足当前规模宏大的大图数据. 在此背景下, 出现了许多经典的分布式大图数据处理系统, 例如 Pregel[8]、GraphX[9]、PowerGraph[10]以及被认为是当前最先进的基于 CPU 的分布式图处理系统 Gemini[11]等.

然而, 与当前先进的基于单机的图处理系统[12−15]相比, 这些经典的分布式图处理系统在处理真实的图数据时并未交付令人满意或稳定的性能优势. 我们深入分析并测试了几个有代表性的开源的分布式图处理系统 (PowerGraph[10]、PowerLyra[16]、Gemini[11]), 总结并归纳出了影响其性能的主要三大挑战.

- 挑战 1: 系统的负载问题

为了在分布式环境中探索图计算的并行性, 系统需要将大图数据划分成不同的子图, 并分配到各个计算节点上, 甚至还可以将子图进一步划分为不同的顶点集并安排到指定的线程上, 以便更好地探索不同计算资源间的并行性. 图划分作为分布式图计算的基础, 它的划分结果一定程度影响着不同计算资源的负载, 负载越多, 所需的计算时间越久. 而当前多数分布式图计算系统普遍认可并采用 BSP (bulk synchronous parallel)作为系统的计算模型[17], 这就使得整个集群的性能取决于最慢的计算资源. 为了保证负载均衡, 现有的分布式系统尝试从图划分层面解决负载问题. 划分方式主要分为两类: 边切的划分方式(edge-cut)[8,18,19]和点切(vertex-cut)[10,16,20]的划分方式. 其中, 边切的划分方式需要保证顶点及其相关数据被划分到同一个计算节点上, 然后基于该条件在计算节点间均分顶点数据, 达到点中心的负载均衡; 而点切的方式则是通过先在节点间尽可能均匀地划分边数据, 然后再采用顶点复制的方式来安排顶点, 达到边中心的负载均衡. 点切的方式很大程度地优化了现实中密律图的划分困境.

然而, 首先, 由于图数据结构的复杂多样性, 使得不同的图数据结构需要采用不同的划分方式来解决负载问题. 而当前图处理系统的单一划分方式并不足以解决图系统的负载难题, 例如, Gemini 采用边切的划分

方式，使得它在处理顶点度极度偏斜的幂律图时，计算节点端的负载极其不均，导致了性能的下降；而 PowerGraph 和 PowerLyra 则采用点切的划分方式，虽然能够均衡地划分幂律图，优化了节点的负载，但点切的划分方式使得它们在处理非幂律图时带来了大量的顶点复制消耗，降低了系统性能．此外，点切的划分方式还会破坏真实大图数据固有的局部性．其次，图计算还有静态不可预测性[21]，即无法通过对静态数据集预测,得出迭代过程中分配给每个算节点的实际负载，这使得图划分并不能完全保证图计算过程中的负载均衡，包括节点间的负载均衡和节点内的负载均衡．综上所述，由于缺乏节点间动态的负载均衡机制，图数据的复杂多样性和图计算的静态不可预测性，使得分布式图计算系统的负载均衡面临着挑战．

- 挑战 2: 特定的通信模型

由于图数据的复杂结构，使得图计算通常在分布式系统中表现为通信密集型[22]．相比普通的计算密集型应用，图计算的集群需要拥有更大的内存容量和更高效的通信机制才能获得更好的性能收益．现有的图处理系统通常以 MPI (message passing interface)或 TCP 为基础来搭建自己的通信接口，相比 TCP, MPI 被证明拥有更好的性能优势[23]，因此，包括本文用于分析测试的 3 个系统在内绝大多数图处理系统都选择采用了 MPI 作为自身的通信基础．然而, MPI 并不是图处理系统理想的通信模型，原因如下．

(1) 随着多核处理器的快速发展出现，有效的图处理系统需要采用异构的并行模型，即计算节点内部采用基于共享内存的并行(如 OpenMP)，而计算节点之间采用基于消息传递的并行，这样可以充分发挥多核处理器的性能，如图 1(a)所示，发送端的每个线程在处理需要发送的数据时，首先将内存中的结构化数据执行序列化操作，打包为 local_buffer，最后调用各自的发送接口执行消息传递，接收端同理．在异构的并行模型下，通信任务同计算任务一样，可以被分发给不同的线程独立完成，这使得每个线程可以完整地贯穿当前任务的整个生命周期，避免了与其他线程交互带来的性能损耗，如原子操作．但对于 MPI 而言，由于固有的设计，使得 MPI 需要采用单线程调用来保证性能优势[24]，MPI 这种固有的漏斗模型导致它在 OpenMP 代码段中只能被串行执行，如图 1(b)所示，由于 MPI 的 send 接口需要被专用线程调用，使得发送端每个线程需要将自身 local_buffer 先通过 gather 操作追加到统一的 send_buffer 内，再由专用的发送线程调用 MPI 的 send 接口执行消息传递：在接收端，专用的接收线程需要事先将收到的消息存储到 receive_buffer 中，再通过 scatter 操作分发给其余线程，最后由各个线程并行处理获取到的计算任务．从 MPI 的漏斗模型中不难发现：首先，无论是发送端还是接收端，OpenMP 的代码段中都被插入了 gather 或 scatter 这样的串行操作，极大地损伤了节点内线程的并行性；其次，由于节点间的消息传递也只能由单线程执行，导致了网络带宽的不充分利用；最后，每个计算节点都需要分配专用的线程负责执行发送和接收，造成了计算资源的浪费，如：用于发送和接收的线程不会参与图处理过程中除通信外的其余工作．因此，相比异构的并行模型，MPI 的漏斗模型会导致图处理系统的性能下降．

(2) 由于 MPI 不是图计算系统专门的通信模型，因此 MPI 的一些语义并不适用于图应用，如: MPI 拥有严格的消息排序要求，而 MPI 的这种排序要求并不能提升图计算系统的性能，反而会影响并行通信时的消息传输速率[23]．

综上所述，分布式图处理系统的性能需要高效的通信模型的支撑，而目前通用的 MPI 并不是图处理系统理想的选择，因此，为分布式图处理系统研究特定的通信机制面临着挑战(注：当前的 MPI 版本虽然已经支持多线程处理，提供了 MPI_THREAD_MULTIPLE 模型，但许多研究已经证明，使用该模型会极大地损伤 MPI 的整体性能[23−25]，因此不被采用)．

- 挑战 3: 轻量级的系统设计

系统的可扩张性是评估系统的一个重要指标，这使得有的系统花费大量的计算资源来维护自身的可扩展性，导致系统的系统指令集和内存引用增大, CPU 的利用率下降，系统的有效性降低．以 PowerGraph 为例，相比当前最先进的基于单机的图处理系统 Ligra[13], PowerGraph 拥有 10 倍的指令和 6 倍的内存引用，但 CPU 利用率却仅有单机图处理系统的 70%[11]．导致这种现象的原因有许多，如维护 GAS 抽象、顶点 ID 的重排序、

全局顶点 ID 与局部顶点 ID 的转换等. 这些支撑可扩展性的操作, 都会对 PowerGraph 的有效性造成影响. 然而对于系统而言, 系统的可扩张性和有效性同等重要, 如何从系统设计的角度权衡两者间的关系具有挑战性.

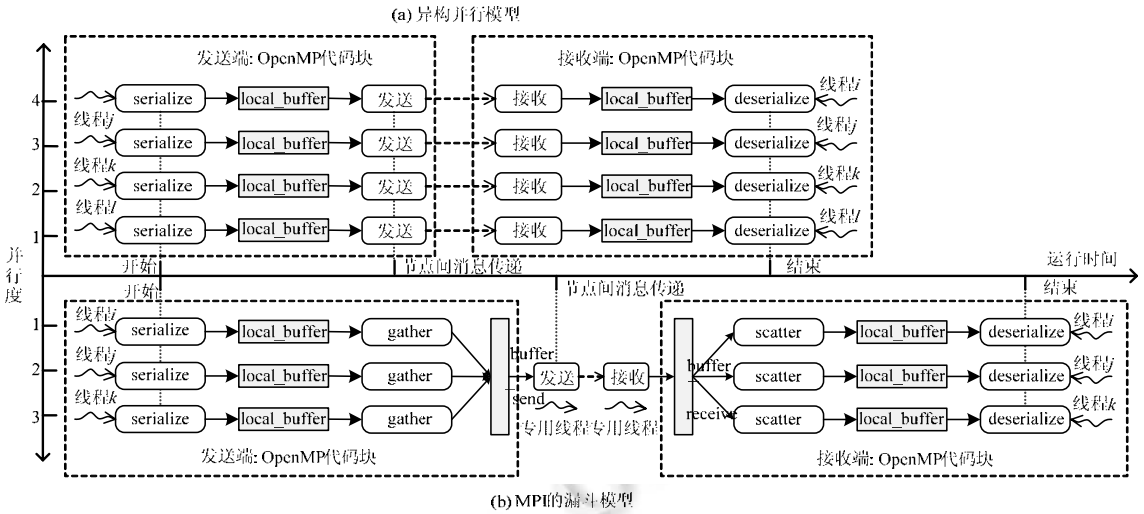


图 1 两种不同的通信模型

为了解决以上三大挑战, 本文提出了 RGraph, 一个基于 RDMA 的高效分布式大图数据处理系统. RGraph 旨在通过充分利用 RDMA 的优势, 来提升图处理系统多个方面的性能. 具体而言, 本文的贡献点如下:

- RGraph 提出了新的基于块的图划分策略. RGraph 的图划分策略从顶点和边的混合角度出发来综合考虑分布式图计算的均衡问题, 不仅能够避免破坏原始图数据的数据局部性, 保证顶点的高效访问; 还能从图划分角度提高节点间的负载均衡.
- RGraph 为图计算编写了基于 RDMA 的高效通信模型. RGraph 的通信模型采用 *WRITE* 操作实现节点间的消息传递, 采用 *READ* 操作去实现远程无感知的任务调度. 为了进一步优化通信模型, RGraph 提出了基于可复用结构的 RDMA 内存优化和基于消息批传输的通信优化. RGraph 还提供了与 MPI 相似的调用接口, 隐藏了复杂的 RDMA 通信逻辑, 使得未接触过 RDMA 的用户也可以轻松调用 RDMA 来实现高效的通信性能.
- RGraph 引入了动态的负载均衡技术, 包括基于任务抢夺的计算节点内线程间的负载均衡和基于任务迁移的计算节点间的负载均衡, 来共同保证图计算的负载均衡. 对于计算节点的负载均衡, RGraph 还通过设计迁移缓存来避免频繁的远程内存读取带来的性能损失, 保证任务迁移的高效性.
- RGraph 为用户提供了轻量级的程序接口, 接口隐藏了复杂的分布式处理逻辑, 用户只需要指定合适的数据类型, 并根据想要实现的算法逻辑编写少量的程序代码, 就可以轻松地在 RGraph 高效地运行自定义的大图算法.
- 大量的实验评估. 我们利用 5 个真实大图数据集和 1 个合成数据集, 在拥有 8 个计算节点的高性能集群上对 RGraph 进行了多方面的性能测试, 包括运行时间、内存消耗、可扩展性等. 实验结果表明, 与当前最先进的基于 CPU 的分布式图处理系统相比, RGraph 平均拥有 3.7 倍的性能提升; 且在极度偏斜的幂律图上, 仍能保证稳定的性能优势.

本文前言部分给出了现有分布式系统面临的挑战, 并提出了准备研究的分布式图处理系统: RGraph. 第 1 节介绍本文的研究背景以及动机. 第 2 节给出 RGraph 的总览, 包括架构、更新模型和核心 API. 第 3 节提出 RGraph 的图划分方案. 第 4 节提出 RGraph 的通信模型, 并针对基本通信模型给出两种不同类型的优化. 第 5 节提出 RGraph 的负载均衡机制以及基于迁移缓存的迁移优化方案. 第 6 节详细地对 RGraph 的各个方面进行

评估. 第 7 节为相关工作. 第 8 节对本文进行了总结.

1 背景与动机

1.1 背景: 图计算

给定一个图 $G=(V,E)$, V 表示 G 中顶点的集合, E 表示 G 中边的集合, 且 G 中的总顶点数和总边数分别表示为$|V|$和$|E|$. 此外, 对于图 G 中的任意顶点 u, 我们使用 $N_O(u)$表示顶点 u 所有的出边邻居, 使用 $N_I(u)$表示顶点 u 所有的入边邻居, 即 $N_O(u)=\{v|(u,v)\in E(G)\}$, $N_I(u)=\{v|(v,u)\in E(G)\}$. 顶点 u 的出度、入度和度分别表示为: $d_O(u)=|N_O(u)|$, $d_I(u)=|N_I(u)|$和 $du=d_O(u)+d_I(u)$. 如果没有特殊说明, 本文讨论的图全部为有向图. 对于无向图, 可以通过将每条边替换为一对有向边进行转换.

现有的图处理系统中, 一般采用基于顶点程序(vertex-program)的图并行计算模型[8−11]. 顶点程序是一组用户自定义的与图算法相关的操作, 它在图算法的迭代过程中用于更新活跃顶点的状态. 在本文中, 图 G 中每个顶点 u 的状态会在算法运行前被初始化为特定的状态. 如图 2(b)所示, WCC 算法的顶点状态会被初始化为自身的 ID, 而 PR 算法的初始顶点状态会被初始化为$(1/|V|)$. 顶点的状态会在随后的算法迭代过程中不断被更新, 直到达到全局的终止条件.

顶点程序可以有两种不同的实现模型,分别是 *Push* 模型和 *Pull* 模型.

- *Push* 模型中, 顶点程序会首先读取 worklist 中顶点的状态, 然后沿着该顶点的出边, 条件更新它的各个出边邻居. 如图 2(c)所示, 顶点 0 会根据当前状态去更新出边邻居 1, 4, 7 的状态.
- 而在 *Pull* 模型中, 顶点程序首先会沿着 worklist 中顶点的入边来读取该顶点所有入边邻居的顶点状态, 然后条件的更新自身的顶点状态, 如图 2 所示, 顶点 0 的状态会依据入边邻居 6, 7 的当前状态进行更新.

RGraph 采用 *Push* 模型作为顶点状态的更新模型.

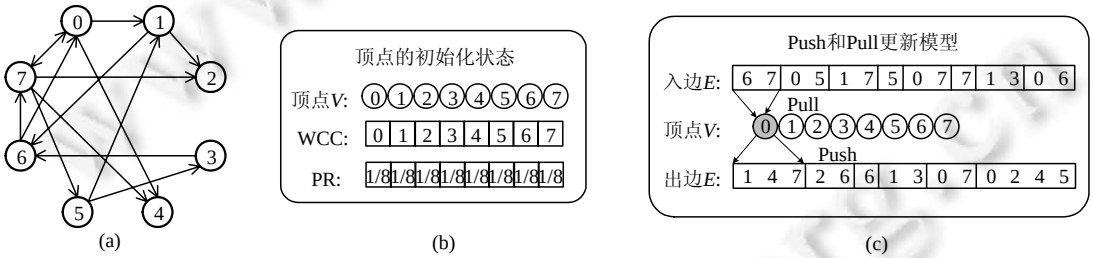


图 2 图计算例子

在图计算方面, 分布式图处理系统与基于单机系统最大的区别是: 分布式系统需要将顶点划分到不同的计算节点上, 并采用消息传递的方式来更新顶点状态; 而单机系统则采用共享内存的方式更新. 因此, 基于单机的图处理系统又被称为基于共享内存的图处理系统. 本文采用经典的 Master-Mirror 表示[10]来同步分布式图处理系统中的顶点状态.

图 3 给出了一个图划分的例子, 原图采用边切(出边)的方式生成图 3(b)所示的划分结果, 顶点$\{A,C,E,G,I\}$被划分到计算节点 1 上, 其余顶点被划分到计算节点 2 上. 每个计算节点都会将划分到的本节点上的顶点定义为 Master, Master 顶点代表着对应顶点的最终状态. 对于源顶点和目的顶点被划分到不同计算节点的边, 如图 3(a)中的(E,B)边, 相关的计算节点会为这些边生成一个新的复制顶点作为该边的目的顶点, 该复制顶点称为 Mirror, 如图 3(b)中所有被灰色填充的顶点, Mirror 将作为 Master 在其余计算节点上的缓存. 通过 Master-Mirror 表示, 分布式图处理系统的通信可以描述为两个不同的阶段.

- 第 1 个阶段为 Reduce: 所有的 Mirror 将自身的顶点状态发送到拥有对应 Master 顶点的计算节点上, 并使用 Reduce 操作在该节点上进行整合.

- 第 2 个阶段为 Broadcast: 所有的 Master 将自身的顶点状态广播给它的所有 Mirror.

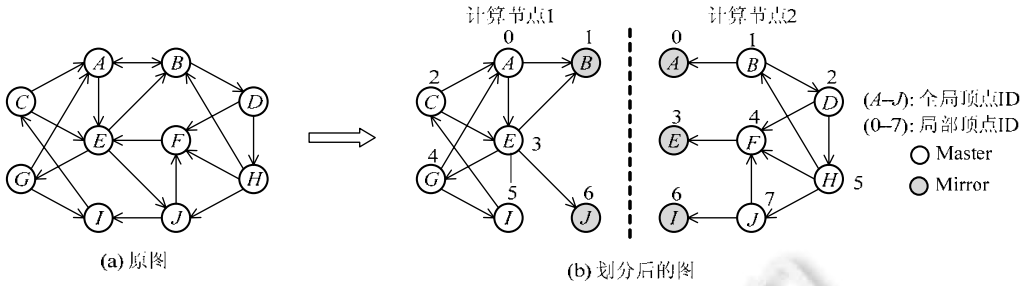


图 3 图划分例子

1.2 背景: InfiniBand和RDMA

InfiniBand (缩写为 IB)是一个用于高性能计算的计算机网络通信标准, 是新一代服务器硬件平台的 I/O 标准. InfiniBand 具有高带宽、低延时、高可扩展性的特点, 近几年被应用于服务器与服务器(比如复制、分布式工作等)、服务器和存储设备(比如 SAN 和直接存储附件)以及服务器和网络之间(比如 LAN、WANs 和 the Internet)的通信.

RDMA (remote direct memory access)是一种直接内存内存访问技术, 其核心功能是允许当前机器直接访问远端机器的内存, 而不需要远端 CPU 的参与, 从而实现高性能通信. RDMA 支持两种单边内存操作语义, 分别是: *RDMA WRITE* 和 *RDMA READ*. *RDMA WRITE* 允许本地计算机在无须远端计算机参与的情况下, 将消息直接写入远端机器的内存中; 而 *RDMA READ* 使得本计算机可以绕开远端计算的 CPU, 直接从远端机器的内存中读取想要的数据. 目前, 支持 RDMA 的网络协议有 3 种, 分别是:

(1) InfiniBand: 即上文提到的新一代网络协议, 本身支持 RDMA, 能够最大程度地保证 RDMA 的性能, 但它需要专用的网卡和交换机.

(2) RoCE (RDMA over converged ethernet): 允许在标准的以太网基础架构(交换机)上使用 RDMA, RDMA 性能不如 InfiniBand, 同时需要支持 RoCE 的特殊网卡.

(3) iWARP (RDMA over TCP): 允许通过 TCP 执行 RDMA 的网络协议, 但 RDMA 性能不如 InfiniBand, 网卡一般为专用的 iWARP 网卡, 如果是普通的网卡, iWARP 栈将在软件中实现, 但会丢失了 RDMA 的大部分性能优势.

本文的集群采用的是基于 InfiniBand 的网络协议来实现 RDMA, 因此我们的集群配备有专用的 IB 交换机和网卡. 总体而言, RDMA 主要有如下优势.

- 零拷贝(zero-copy). RMDA 使得应用程序在不涉及网络软件栈的情况下直接执行数据传输, 需要传输的数据可以直接与缓冲区交互, 而不需要被复制到网络层.
- 内核旁路(kernel bypass). RMDA 使得应用程序可以直接在用户态执行数据传输, 而不需要在内核态与用户态之间做上下文切换.
- 无须 CPU 干预. RMDA 使得应用程序可以直接访问远端主机的内存而不需要远端 CPU 的任何参与.

需要强调的是: RDMA 仍处于发展阶段, 基于 InfiniBand 的 RDMA 业界缺乏便捷的编程接口, 而官方只提供了基本的底层通信接口 IB verbs. 这使得程序员需要基于这些通信接口从底层开始搭建自己的 RDMA 通信模型, 如 QP 队列、完成队列、内存注册、保护域等的创建、修改和销毁, 以及正确处理与 QP 相关的工作请求等.

1.3 动 机

通过前言中提到的挑战可知, 图计算通常在分布式系统中表现为通信密集型, 用于图计算的集群需要拥有更高效的通信机制才能获得更好的性能收益. 而现有的图系统采用的基于 MPI 的漏斗型通信模型只能为图

计算系统提供次优的性能收益. 采用基于 RDMA 的通信模型来替换 MPI, 是分布式图处理系统更好的选择. 原因如下.

(1) 通信密集型的分布式图计算系统需要高带宽低延迟的通信组件来保证系统的性能. 相比于 MPI, RDMA 的零拷贝、内核旁路和新的通信标准, 使得 RDMA 拥有更高的带宽和更低延迟, 能够为分布式图计算系统带来显著的通信收益.

(2) RDMA 无须远端 CPU 干预的特点, 能够很好地满足分布式图计算系统的需求:

- 首先, 基于 RDMA 的通信协议使得计算节点在通信时可以节省更多的 CPU 资源, 增大了计算时间重叠通信时间的机会.
- 其次, RDMA 的单边 *READ* 能够很好地解决图处理系统负载不均的挑战: 对于负载较大的计算节点, 节点上所有计算资源处于忙碌状态, 而 *READ* 操作使得剩余的计算节点可以独立地从忙碌的节点上分摊到计算任务来平衡集群的负载(原先节点无须分配任何计算资源去参与调度), 从而保证任务迁移的同时, 本地计算资源保持着最大利用率, 这是 RDMA 独有的优势.

(3) 相比于为应用程序人员开发的 MPI, RDMA 的通信接口 IB verbs 库提供更底层的程序接口, 这使得研究人员能够更有效地研究特定的符合分布式图处理系统的通信模型, 如异构并行模型.

综上所示, RDMA 为本文提供了研发高效图计算系统的机会, 本文将其命名为 RGraph, 一个基于 RDMA 的高效分布式大图数据处理系统.

2 RGraph 总览

2.1 RGraph的架构

RGraph 的架构总览如图 4 所示.

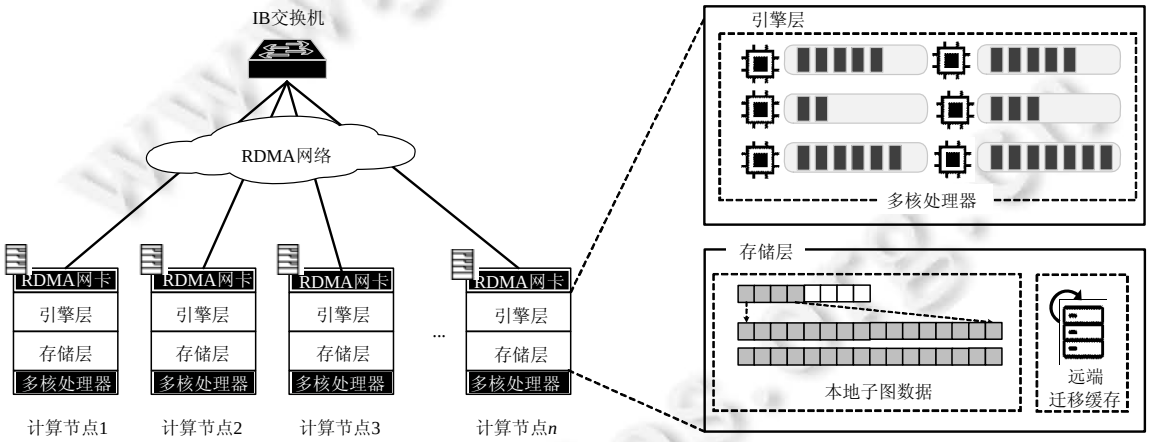


图 4 RGraph 系统架构总览

RGraph 是一个基于 RDMA 的分布式内存处理系统, 它遵循去中心化的体系结构, 参与计算的所有计算节点都通过高带宽、低延迟的 RDMA 网络连接到 IB 交换机上, 一起构成高效能集群. 集群中每个机器都配备一张 RDMA 网卡、2 个多核处理器以及 128 GB 的 DRAM (dynamic random access memory). 为了保证性能, RGraph 在每个机器上仅运行 1 个计算节点, 每个计算节点可以分为两层: 引擎层和存储层.

- 引擎层. 该层采用 Worker-Thread 模型, 将多核处理器的 N 个核分配给 N 个不同的工作线程, 每个工作线程又通过绑定一个工作队列来连续执行图任务. RGraph 的工作线程主要执行两种类型的计算任务: 本节点的图计算任务和远程节点的迁移任务. 单次迭代计算开始后, 所有工作线程并行处理本节点的图计算任务, 并采用任务抢夺的方式来保证节点内的负载均衡. 当本节点的所有任务完成后,

所有工作线程转为参与远程节点的迁移任务, 并行地从远端节点中获取计算任务并处理, 保证节点间的负载均衡.

- 存储层. 如图 4 所示, 存储层可以分为两大类: 存放划分后子图数据的内存和存储远程迁移数据的缓存. 其中,
  - 内存中的子图数据采用经典的 CSR (compressed sparse row)作为基本的存储结构. 为了进一步优化内存的访问效率, RGraph 最终采用 CSR+Bitmap 的数据结构. Bitmap 用于快速裁剪无须本地计算的顶点(如无邻居或邻居不在本地内存中的顶点), 同时, Bitmap 占用较小内存的优势, 使它很容易被缓存到现代处理器较大的最后一节缓存中, 以加速处理器的访问速率. 此外, 所有计算节点内存中的子图数据都支持远端节点的直接访问.
  - 远程数据缓存用于存储任务迁移过程中从其余计算节点获取的图数据,这些数据根据指定的数据结构和规则进行缓存,来保证任务迁移的执行效率.

2.2 RGraph的更新模型

同大多数分布式图处理系统一样, RGraph 采用 BSP 作为集群的计算模型, 采用 Push 模型作为顶点状态的更新模型. 如图 5 所示, 在连续的两次迭代中, 需要显式地插入一次 barrier 操作, 保证计算节点间的同步. 在每次迭代中, 计算节点端的每个 master: 首先, 根据用户自定义的顶点程序, 沿着出边来更新自己的邻居顶点; 然后, 被更新的邻居顶点中, mirror 会将它的顶点状态以消息的形式传递给对应的 master; 最后, master 会采用用户自定义的 reduce 操作来获取最终的顶点状态. 集群的迭代计算会不断执行, 直到用户自定义的终止状态达到或图中所有的顶点状态不再变化为止.

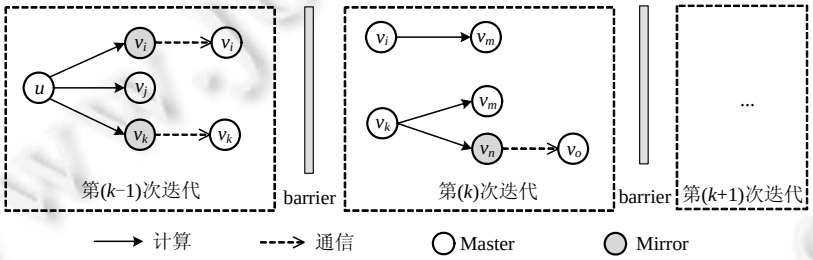


图 5 RGraph 更新模型总览

2.3 RGraph的核心API

RGraph 采用 C++语言编写, 共包含 34 个程序文件, 共计 7 800 行代码. RGraph 的程序接口隐藏了复杂的分布式处理逻辑, 用户只需要指定合适的数据类型, 并根据相应的算法逻辑编写少量的程序代码, 就可以轻松地在 RGraph 高效地运行自定义的大图算法. RGraph 在保证轻量级的系统设计的同时, 主要提供以下程序接口.

1. *loadGraph*(*graphPath*): *RG_ingress*. 用户给定准备运行的图文件, *loadGraph* 就可以为每个计算节点生成必要的上下文, 如图数据的划分信息、子图数据的引用等, 上下文保存在 *RG_ingress* 中.
2. *processVertice*(*RG_ingress*,*work*,*actine*):*R*. 对于给定的上下文 *RG_ingress*, *processVertice* 应用用户自定义的函数 *work* 处理 *actine* 中被标记的顶点, 表示为 $R=\{u\in active\mid\sum work(u)\}$.
3. *processEdge*(*RG_ingress*,*compute*,*reduce*,*active*): *R*. 对于给定的上下文 *RG_ingress*, *processEdge* 应用用户自定义的函数 *compute* 处理 *active* 中被标记顶点的所有出边, 并将更新的 mirror 顶点根据用户自定义的 *reduce* 操作, 整合到 master 端. 表示为:

$$E_{active}=\{(u,v)\mid u\in active\},\ R=\{(u,v)\in Eactive\mid\sum compute(u,v)\cup reduce(v)\}.$$

本文以 SSSP 算法为例, 说明 RGraph API 的使用. 如算法 1 所示, 用户首先需要对 SSSP 执行初始化操作

(第 1 行–第 6 行), 包括初始化 SSSP 所需的顶点状态数组 *distance*、初始化存储当前激活顶点的 Bitmap 结构 *activeIn*. 其中, *activeVertice* 记录下一轮的迭代中激活的顶点个数, 用于判断是否达到算法的终止状态. 最后, 主函数循环调用 *processEdge*(第 7 行–第 11 行), 直至 *activeVertice*=0, 算法结束. 算法 1 还给出了 SSSP 算法顶点程序的实现逻辑(第 13 行–第 31 行), 该部分由用户提供.

算法 1. SSSP.

```
1.  RG_ingress rg_ingress=loadGraph(graphPath);   //获取计算节点的上下文
2.  Distance=[∞,∞,∞,…,∞];   //初始化顶点状态
3.  Distance=[source]=0;
4.  activeIn.setBit(source);
5.  pushToWorkList(source);   //初始化 worklist
6.  activeVertices=1;
7.  While activeVertices>0 do   //所有顶点状态稳定后, 退出循环
8.     activeOut.clear(·);
9.     activeVertices=processEdge(rg_ingress,sssp_compute,sssp_reduce,activeIn);
10.    std::swap(activeIn,activeOut);
11. End While
12.
13. sssp_compute(src,nbrIterator) do
14.    activated=0;
15.    While nbrIterator.hasNext(·) do
16.       nbr=nbrIterator.next(·);   //获取邻居
17.       cur=distance[src]+nbr.edgeData
18.       If atomicMin(&distance[nbr.dst],cur)   //更新为最小值
19.          activeOut.setBit(nbr.dst);
20.          activated=activated+1;
21.       End If
22.    End While
23. End
24.
25. sssp_reduce(mirror,msg) begin
26.    If atomicMin(&distance[mirror],msg)   //整合为最小值
27.       activeOut.setBit(mirror);
28.       return 1;
29.    End If
30.    return 0;
31. End
```

3 RGraph 的图划分

RGraph 采用基于块的划分方式来对大图数据进行划分. 具体而言, 对于拥有 P 个计算节点的集群, 全局图 $G=(V,E)$中的顶点集 V 被划分为 P 份不相交且顶点 ID 连续的子集$(V_0,V_1,V_2,\ldots,V_p)$, 每份顶点子集被安排到一个计算节点上, 每个计算节点拥有的边子集定义为 $E_i=\{(u,v)\in E|u\in V_i\}$, 即所有顶点的出边都会跟随顶点被划分到相同的计算节点上. 图 3 给出了 RGraph 的图划分实例. RGraph 采用基于块划分的方式主要有如下原因.

- 满足 RGraph 轻量级的设计初衷. 基于块的大图划分能够避免顶点的重排序以及全局顶点和局部顶点的映射, 有效地减少系统所需的指令量.
- 保护真实图数据的局部性. 真实的大图数据本身具有数据局部性, 如 Facebook 根据地理位置来生成社交网络, Webgraph 的 URL 排序规则使得相邻的 ID 尽可能地互联. 基于块的大图划分能够有效地避免破坏已有的局部性, 保证顶点的高效访问.
- 减少通信量. 基于块的大图划分使得每个顶点的全部出边都跟随顶点被划分到相同的计算节点上, 使得计算节点在同步时, 只需执行 reduce 操作, 而不再需要调用 broadcast 操作.

基于块划分的最大难点是如何划分顶点集 V, 平分顶点集或平分边集并不是最优的划分方式, 因为现实中大图的顶点度分布并不均衡. 特别是幂律图, 均分顶点或边很容易导致子图数据分配的不均, 导致计算节点的负载均衡问题. 为了解决以上难题, RGraph 参考了 Gemini 的解决思想, RGarph 顶点数组的划分采用公式(1)所示的平衡函数.

$$F(P_i)=\alpha\cdot|V_i-avg_{vertex}|+|E_i-avg_{edge}| \tag{1}$$

其中, α平衡系数, RGraph默认定义为$\lfloor|E|/|V|\rfloor$; $avg_{vertex}=|V|/P$, $avg_{edge}=|E|/P$表示图的平均顶点数和边数. 公式(1)中的解决方案从顶点和边的混合角度出发来综合考虑图划分的均衡问题, 能够有效地解决图数据的分配不均问题. 相比于 Gemini 的平衡函数, RGraph 的平衡函数加入了对数据集特征的考虑, 拥有更好的平衡优势. 但该方案依然不能稳定地保证图计算的负载均衡, 直观的原因是α不具有通用性, 其次还有前言部分提到的图的静态不可预测性. 然而不同于 Gemini, RGraph 拥有动态的负载调节机制, 降低了系统对图划分策略的依赖. 在图计算的过程中, 负载调节机制帮助 RGraph 实时调节计算节点间的负载, 能够有效地降低负载不均带来的性能影响, 本文将在第 5 节详细介绍 RGraph 的负载均衡.

本文以图 2(a)给出了 RGraph 图划分的一个例子. 为了直观地表示, 本例中我们假设集群中共有 3 个计算节点, P=3, 且 a=1.1. 具体划分过程如下.

1. 以 Gemini 的划分结果作为初始输入: 此时的顶点划分数组为[0,2,4,8], 3 个计算节点分别拿到的顶点个数为 2, 2, 4. 计算节点 1 负责顶点 ID 为 0–1 的顶点, 计算节点 2 负责顶点 ID 为 2–3 的顶点, 计算节点 3 负责顶点 ID 为 4–7 的顶点.
2. 寻找集群中 $F(P_i)$最大的计算节点: *maxNodeId*, 此时有 $F(P_1)$=1.06, $F(P_2)$=4.39, $F(P_3)$=4.8. 所以 *maxNodeId*=3, 且 $\sum_1^n F(P_i)=10.25$.
3. 寻找 *maxNodeId* 的邻居计算节点中 $F(P_i)$最大的计算节点: *nbrNodeId*, 计算节点 3 的邻居计算节点为节点 2, 所以 *nbrNodeId*=2.
4. 将 *maxNodeId* 和 *nbrNodeId* 作为本次调节的两个计算节点: *nodeLeft* 和 *nodeRight*, 开始调节. 本例中, *nodeLeft*=2, *nodeRight*=3.
5. 根据 *nodeLeft* 和 *nodeRight* 中实际存储的顶点数和边数, 判断顶点划分数组的变化. 本例中, 机器 3 拿的边和顶点都比机器 2 多, 所以顶点划分数组中的位置 3 处对应的数应该增大.
6. 利用枚举法寻找 *nodeLeft* 和 *nodeRight* 中的平衡状态, 本例中, 位置 3 处的 n 的取值范围是 5≤n≤7, 当 n=5 时, $\sum_1^n F(P_i)=8.78$; 当 n=6 时, $\sum_1^n F(P_i)=6.18$; 当 n=7 时, $\sum_1^n F(P_i)=6.45$. 最后可知, 当 n=6 时, 本轮中的 $\sum_1^n F(P_i)$ 取得最小值. 所以 本轮的顶点划分数组调节为[0,2,6,8]. 当前步骤虽然采用枚举思想, 但并不会大幅度降低图划分的性能, 因为 n 的取值本身即有一定的范围界定, 如 n 的取值在本例中为 5≤n≤7.
7. 以 *nodeLeft* 和 *nodeRight* 作为初始状态, 对集群中的其他计算节点循环执行最后 4 步, 直到 $\sum_1^n F(P_i)$ 整体达到最小. 本例的下一轮中的 *nodeLeft*=1, *nodeRight*=2, 但此时所有 $\sum_1^n F(P_i)$ 都不会求得比上轮中的 $\sum_1^n F(P_i)$ 更小的值, 所以停止计算, 最终的顶点划分数组为[0,2,6,8].

通过划分, RGraph 的划分方式使得 3 个不同的计算节点划分到了更加平衡的顶点和边数. 具体为: 计算

节点 1 获得 2 顶点和 5 条边(Gemini 为 2 顶点 5 边), 计算节点 2 获得 4 顶点和 3 条边(Gemini 为 2 顶点 1 边), 计算节点 3 获得 2 顶点和 6 条边(Gemini 为 4 顶点 8 边). 因此, RGraph 的划分方式加入了对数据集特征的考虑, 更能保证计算节点间顶点和边的均匀划分.

4 RGraph 的通信模型

RGraph 采用基于内存语义的 RDMA 单边 *WRITE* 和 *READ* 操作来设计自己的通信模型, 旨在充分发挥 RDMA 和当代多核处理器的性能优势. 具体而言, RGraph 采用 *WRITE* 操作实现节点间的消息传递, 采用 *READ* 操作去实现远程无感知的任务调度.

4.1 基本通信模型

图 6 以 2 个计算节点、每个计算节点拥有 3 个线程为例, 给出了 QP(queue pair)连接的建立和基于 QP 连接实现的基本通信模型. 如图 6 的下半部分所示, 建立 QP 连接主要分为 3 步: 第 1 步, 每个计算节点初始化各自的 RDMA, 包括获取设备信息、注册 RDMA 内存、绑定保护域等; 第 2 步, 采用 TCP 连接与其他计算节点交换各自的 RDMA 上下文信息; 第 3 步, 利用获取到的信息与集群中的其他计算节点建立 QP 连接.

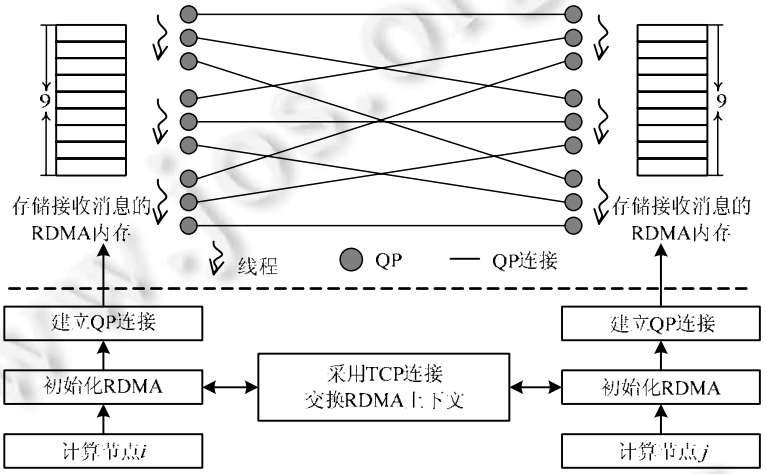


图 6 基本的通信模型

图 6 的上部分展示了基于 QP 连接实现的 RDMA 基本通信模型, 该模型会为集群中计算节点间的每个线程对建立一个单独的 QP 连接, 使得每个线程都能与远端计算节点上的任意线程进行私有通信; 该模型同时为每个 QP 连接预留单独的缓存区来存储接收到的信息, 保证了在利用 *RDMA WRITE* 执行远端数据写入时的无远端 CPU 干预性和数据接收的正确性(注: RGraph 不采用多个 QP 连接使用同一个缓存区, 原因是使用统一缓冲区需要分配额外的线程来协调远端内存的写入地址, 该方式不仅会减少工作线程的数量, 还会降低工作线程的并行性).

4.2 RDMA内存优化

基本通信模型虽然满足了在无 CPU 干预的情况下, 集群中的任意线程之间都可以进行私有通信, 实现了并行的消息传递, 但随着集群中计算节点的增加, 基本通信模型的 QP 连接方式会造成内存资源的浪费, 影响系统的性能. 这是因为为了保证数据写入的正确性, 系统需要为每一个 QP 连接分配单独的数据缓冲区, 用于存储接收到的消息. 从图 6 不难推断出: 对于一个拥有 m 个计算节点、每个计算节点拥有 t 个线程的集群, 基本通信模型一共需要建立$(m-1)\times t^2$ 个 QP 连接, 如果缓冲区的大小为 k, 那么每个计算节点至少需要预留 $k\times(m-1)\times t^2$ 大小的内存用于存储接收到的消息, 这无疑增大计算节点的内存压力. 为了缓解由于使用基本通信模型造成的内存压力, RGraph 结合图计算系统的特点提出了两种优化方法.

- 方法 1: 减少 QP 的连接数. 在图处理系统中, 顶点的状态可以被本节点的所有线程访问, 为此, 我们会选择将顶点的状态数组交叉地(interleave)存储到内存中, 因此, 顶点状态消息并没有固定的接收规则, 如: 不存在固定的顶点需要被固定的线程接收并处理的情况. 因此, RGraph 采用线程与机器配对的方式来建立 QP 连接, 该方法不仅能够实现并行的消息传递, 还能将所需的 QP 连接数降低到 $(m-1)\times t$ 个, 此时, 每个计算节点只需要预留 $k\times(m-1)\times t$ 大小的内存来存储接收到的消息.
- 方法 2: 少申请多复用周期处理. 分布式图计算系统中, 首先, 计算节点间消息传递的最小单元是由顶点 ID 和它的顶点状态组成的结构体, 我们将该结构体定义为最小消息单元(*min_msg*), 最小消息单元一般仅占用 8 字节的空间(前 4 字节存储顶点 ID, 后 4 字节存储顶点状态), 因此线程单次发送的消息块不会太大(注: 消息块是第 4.3 节 RGraph 为了优化传输引入的定义, 多个最小消息单元会被合并为一个消息块执行统一发送, 消息块的大小会有具体的限定, 同样不会太大, 如 RGraph 默认限定为 2 048 字节); 其次, 图计算过程的不可预测性, 使得系统无法事先预测每个线程的实际总接收量, 该原因导致缓冲区申请太大会造成内存的浪费, 申请太小会造成数据覆盖, 引发错误. 为此, 我们选择“少申请+多复用+周期处理”的方案进行内存优化.

如图 7 所示, 我们将每个 QP 的接收缓存区定义为大小为 R 字节的复用结构, RGraph 中 R 默认为 2 097 152, 即 2 MB. 该结构一共由 4 块区域组成, 分别存储了接收端已处理的消息、远端已写入的消息、远端正在写入的消息和发送端已知的可用空间. 这些区域由 3 个不同的指针进行定界, 指针按照指定的规则协同移动, 保证复用结构的高效运行. 本文以 3 个指针为中心, 解释可复用结构的工作原理如下.

- 发送端写入指针 P_{send_write}: 该指针存储在发送端, 用于标记发送端给接收端写入消息数据时的内存地址. 当消息被成功写入接收端的缓冲区内时, 发送端会主动将发送端写入指针后移到消息尾部, 如图 7 中的过程 3 所示.
- 接收端处理指针 $P_{receive\_deal}$: 该指针存储在接收端, 用于标记接收端拿取消息的内存地址. 此外, 为了使接收端能够正确地确定消息的长度, 本文将消息定义为 2 部分: 数据头 msg_{head} 和数据域 $P_{context}$ 和数据尾 msg_{tail}, 数据头固定占用 4 字节, 保存了该消息中的最小消息单元个数($|min\_msg|$); 而数据域保存了实际的消息数据; 数据尾同样占用 4 字节, 用于验证消息写入的完整性(当数据尾存储的数据和数据头相同时, 该消息才能被接收端处理). 由于最小消息单元由固定的字节组成(如 8 字节), 因此, 接收端可以根据公式(2)推算出当前消息的尾部位置:

$$\text{尾地址}=(P_{receive\_deal}+sizeof(min\_msg)+sizeof(min\_msg)\times|min\_msg|)\%R \tag{2}$$

  当消息被处理后, 接收端会主动将接收端处理指针移动到消息尾部, 如图 7 中的过程 2 所示. 需要强调的是: 在接收端, 线程与要处理的复用缓冲区是一对多的关系. 换言之, 任何一个复用缓冲区只能由一个线程处理, RGraph 的处理逻辑保证了不会出现同一计算节点多个线程并发访问同一可复用缓冲区的情况.
- 发送端终止指针 $P_{send\_write\_end}$: 该指针存储在发送端, 用于判断缓存区的剩余可用空间, 可用空间如公式(3)所示.

$$\text{可用空间}=\begin{cases}P_{send\_write\_end}-P_{send\_write}, & \text{若} P_{send\_write}\leqslant P_{send\_write\_end}\\ R+P_{send\_write\_end}-P_{send\_write}, & \text{若} P_{send_{write}}>P_{send\_write\_end}\end{cases} \tag{3}$$

  只有当可用空间大于当前准备发送的消息大小时, 才会执行 *WRITE* 操作. 由于发送端无法即时获取到接收端消息处理的具体时间, 因此发送端终止指针需要由接收端主动更新. 为了减少更新消耗, 接收端每处理完缓冲区总容量一半的消息数据后, 如 1 MB, 会主动发送一段更新消息, 更新消息将发送端终止指针移动到当前接收端处理指针所在的位置, 如图 7 的过程 1 所示.

综上所述, 在发送端, 发送端终止指针负责判断是否有足够的空间执行远端消息写入, 如果空间足够, 则调用发送端写入指针执行消息写入; 在接收端, 接收端处理指针负责接收消息的处理以及发送端终止指针的更新, 3 个指针协同工作, 保证 RGrpah 的通信性能. 复用结构需要在接收端执行周期处理, 如果消息在缓冲区

内堆积太多，会造成可用空间的不足，使得发送端无法执行写入操作. RGraph 默认采用“线程每处理 100 个工作任务后，转而执行一次消息处理”的规则来执行周期处理. 我们对 RGraph 的复用结构进行了大量的实验测试,实验结果显示：当前的默认配置及规则可以完全满足所有数据集下的消息传递需求，同时还具有极高的内存复用率(如：在数据集 Twitter2010 下执行 PR 算法，总大小为 2 MB 的缓冲区，利用复用结构，线程在单次迭代中，成功接收了 23 MB 的消息数据，复用率达 1150%).

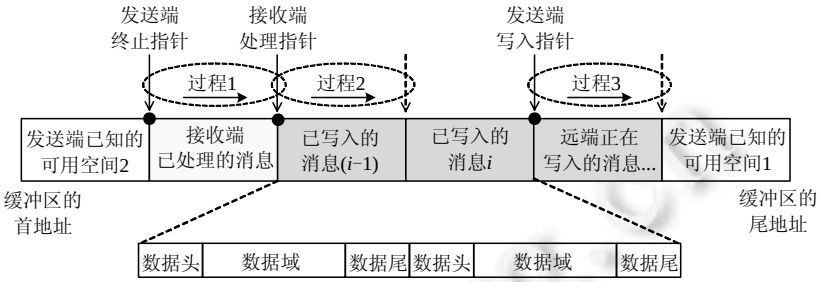


图 7　可复用的缓冲区结构

4.3 消息的批传输优化

RDMA 每次传输数据时都需要经过一个预处理的阶段，从第 4.2 节可知，图处理系统的最小消息单元非常小，如果以最小消息单元为单位执行消息传输，会降低传输的性能. 解决方法是：先在发送端执行 Gather 操作，将多个最小消息单元合并为消息块，然后再以消息块为单位调用 *WRITE* 操作执行消息写入. 不同于 MPI, RGraph 的 Gather 操作发生在每个线程内部，操作对象是最小消息单元，线程间的 Gather 操作相互独立;而 MPI 的 Gather 操作发生在线程之间，操作对象是每个线程的 local_buffer，线程间的 Gather 操作串行执行. 算法 2 给出了 RGraph 每个线程的批传输逻辑, send 保证达到以消息块的大小才会调用 *WRITE* 操作, flushSend 一般在通信代码段的最后一步调用，保证发送缓冲区中的数据全部完成发送，其中, *msgBlock* 为发送缓存区.

算法 2. RGraph 的批传输.

```
1.  send (destNode,msg) begin
2.    msgBlock[destNode].oarchive<<msg;   //将发送的消息序列化，并写入 msgBlock 的数据流中
3.    msgBlock[destNode].num=msgBlock.num+1;   //msgBlock 的消息计数器加一
4.    If msgBlock[destNode].num≥MAX_SEND   //MAX_SEND 在 RGraph 中默认为 256
5.      msgBlock[destNode].oarchive.head=MAX_SEND;   //将 msg 的个数写入数据流的首部
6.      rdma_write(destNode,msgBlock[destNode].oarchive);   //使用 RDMA WRITE 执行批传输
7.      msgBlock[destNode].clear(·);   //清空
8.    End If
9.  End
10.
11. flushSend(·) begin
12.   For node:0–nodeNum   //依次检查当前线程的所有 QP 连接发送缓冲区
13.     If msgBlock[node].num>0   //判断发送缓冲区中是否有未发生的消息
14.       msgBlock[node].oarchive.head=msgBlock[node].num
15.       rdma_write(node,msgBlock[node].oarchive);   //使用 RDMA WRITE 执行传输
16.       msgBlock[node].clear(·);   //清空
17.     End If
18.   End For
19. End
```

4.4 RGraph的通信API

为了方便用户自定义图算法, 实现自身的通信逻辑, 我们提供了 RGraph 的通信 API 供用户调用. 如图 8 所示, RGraph 将通信 API 封装到了 *Buffer_exchange_RDMA* 类中, 并提供了与 MPI 相似的调用接口.

```
Class Buffer_exchange_RDMA begin
1.      Buffer_exchange_rdma(threadNum,MAX_SEND=256);   //构造函数
2.
3.      void send(destNode,threadId,sendMsg);
4.      void flushSend(threadId);
5.      bool receive(threadId,recMsg);
6.      void all_reduce(value,operator);
7.      void all_gether(value);
8.      void barrier(·);
9.
10.     void rdma_read(destNode,destAdr,lenght,localAdr);
11.
12.     uint64_t rdma_cas(threadId,destNode,destAdr,compare,swap);
13.     uint64_t rdma_fetch_and_add(threadId,destNode,destAdr,addValue);
14.  End
```

图 8 RGraph 的通信 API

RGraph 的通信 API 可以分为 3 类: (1) 基于 *WRITE* 操作的远程消息写入 API (第 3 行–第 8 行), 用于图计算系统的快速消息传递; (2) 基于 *READ* 操作的数据直接访问 API (第 10 行), 用于图计算系统的任务调度; (3) RDMA 原子操作 API (第 12 行、第 13 行), 保证远端数据访问的原子性.

第 3 行和第 4 行的 *send* 和 *flushSend* 操作由发送端的多线程调用, 实现了并行的消息传递. 第 5 行的 *receive* 操作由接收端的多线程调用, 用于接收由远端节点写入接收缓冲区中的消息. 第 6 行和第 7 行的 *all_reduce* 和 *all_gether* 提供了类似于 MPI 中 MPI_Allreduce 和 MPI_Allgather 的功能. 第 8 行的 *barrier* 由单线程调用,用于集群中所有计算节点的同步, 功能同 MPI 中 MPI_Barrier. 第 10 行的 *rdma_read* 操作支持多线程调用, 用于将 *destNode* 计算节点端 *destAdr* 处, 长度为 *lenght* 的数据写入自己的 *localAdr* 中, 提供了远端节点快速访问本地的内存的方式. 第 12 行、第 13 行为 RDMA 的原子操作, 用于保证远端数据访问的原子性.

RGraph 的通信 API 不仅提供了与 MPI 相似的调用接口, 同时还将 RDMA 的初始化与优化、序列化与反序列化等复杂的程序逻辑全部隐藏到底层, 使得未接触过 RDMA 的用户也可以轻松使用 RDMA 来编写高效的通信代码.

5 RGraph 的负载均衡

由前言部分可知, 图计算具有静态不可预测性, 即事先无法通过静态数据, 如图分割, 保证每次迭代过程中的负载均衡. 为此, RGraph 引入了动态的负载均衡技术, 包括基于任务抢夺的计算节点内线程间的负载均衡和基于任务迁移的计算节点间的负载均衡. 为了表述清晰, 本文将需要任务迁移的计算节点称为高负载节点, 将主动执行迁移任务的计算节点称为迁移节点, 即迁移节点会主动迁移高负载节点上的任务.

5.1 计算节点内的负载均衡

现有的图计算系统大多采用点中心的程序模型, 每个工作线程会采用均分的方式从 worklist 中获取需要处理的顶点作为自身的计算任务. 但由于顶点度的差异, 平分顶点并不能保证线程间的负载均衡. 不同于现有系统采用的任务窃取方式, RGraph 采用任务抢夺的方式来解决节点内的负载均衡问题. 本文采用任务抢夺机制作为节点内的平衡策略, 有 2 点原因: 首先, 任务抢夺方式能够有效解决节点内的负载均衡, 性能与传统的任务窃取相似, 本文在实验部分采用相关实验进行了证明, 见第 6.4.2 节; 其次, 任务抢夺方式还可以更好地适配本文提出的节点间的任务迁移策略, 这是传统任务窃取方式无法实现的. 本文将在下节详细介绍基于任务抢夺方式的节点间任务迁移策略.

任务抢夺机制为 worklist 分配一个共享计数器, 用于标识 worklist 中的处理进度. 为了减少共享计数器的

原子操纵次数, RGraph 将线程的单次任务抢夺数(*min_work*)定义为 64 个顶点, 即每个线程一次从 worklist 中抢夺 64 个顶点依次执行计算(RGraph 的默认 *min_work* 设置为 64, 主要原因是: RGraph 的任务抢夺机制使得分配给线程的任务数只能由该线程单独完成, 因此事先给单个线程分配太大的任务量会降低 RGraph 的负载均衡性能, 且现实中的大图数据顶点度大多分布不均匀, 如呈现幂律分布, 会进一步加剧该现象. 为此, RGraph 同现有其他系统一样, 采用 64 作为线程的单次任务抢夺数, 用户也可以根据 RGraph 的接口自定义该值). 最快完成的线程将优先从 worklist 中抢夺下一批顶点, 直到 worklist 中的所有任务被处理完成为止. 图 9 给出了任务抢夺的一个例子, 假设当前节点内共有 4 个线程参与 worklist 中的任务计算, 当所有线程分配到任务后, 共享计数器移动到图中所示位置, 共享计数器右边为未分配的任务. 虽然每个线程分配的任务中具有相等的顶点数, 如 64 个顶点, 但每个线程的工作量却不同, 主要取决于当前 64 个顶点拥有的总边数, 图中的阴影部分表示每个线程的实际工作量. 可知, 线程 *j* 由于工作量最少, 会最先完成本次计算. 按照任务抢夺机制, 线程 *j* 会最先抢夺到下一批工作任务, 如图 9 中的弧线箭头所示, 所有线程按照上述方式执行抢夺操作, 直到 worklist 中所有任务被完成.

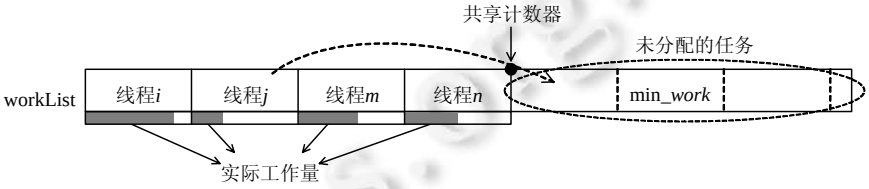


图 9 任务抢夺的例子

5.2 计算节点间的负载均衡

相比于计算节点内的负载均衡, 计算节点间的负载均衡更为复杂, 主要有以下原因: (1) 节点间的负载均衡不仅与顶点度的大小有关, 还与计算节点的 worklist 中的任务数有关; (2) 节点内的负载可以通过共享内存执行交互, 而节点间的负载则需要通过消息传递交互; (3) 节点间的负载均衡不仅需要获取需要迁移的任务, 还需要高效地获取任务所需的原数据, 如任务顶点的邻居数据; (4) 高负载节点希望迁移节点可以分摊自身的任务来减少负载, 但高负载节点上的所有工作线程此时都处于忙碌状态, 没有多余的计算资源可以用于任务迁移, 任务迁移需要由迁移节点独立完成(注: 现有的系统为了实现节点间的负载均衡, 会在每个计算节点上分配专用的迁移线程来执行迁移工作, 但专用线程有以下缺点: 首先, 造成计算资源的浪费,迁移线程不会参与除迁移外的任何工作, 在无迁移任务时, 该线程处于空闲状态; 其次, 单线程的处理速度未必能够同时满足迁移节点上所有线程的迁移请求, 一旦不满足, 就会造成线程的闲置). 为此, 本文提出了基于 RDMA 单边 *READ* 的任务迁移机制, *READ* 操作最大的优势是: 使得迁移节点可以独立地从高负载节点上获取到需要迁移的任务和任务所需的原数据.

5.2.1 基本的任务迁移策略

计算节点间的任务迁移首先需要解决以下问题: (1) 迁移节点如何寻找高负载节点? (2) 高负载节点上的哪些任务需要迁移? (3) 如何高效地获取迁移任务所需的原数据? RGraph 设计了新的数据结构和机制来解决以上问题.

RGraph 完成图划分后, 集群中所有的计算节点都需要根据自身的划分信息维护图 10 中的 nodeAdrMsg 数据结构(第 1 行–第 7 行), 并通过 RGraph 的消息传递接口广播给其他计算节点. NodeAdrMsg 数据结构维护了获取当前计算节点原图数据和任务列表的基本途径, local_subgraph_address 标识了原图数据的存储地址, *worklist_address* 标识了现有任务的存储地址. 通过 nodeAdrMsg 数据结构, 任何节点都可以通过 *READ* 操作在远端节点无感知的情况下获取到任务迁移所需的基本数据.

RGraph 在计算的过程中, 将每次迭代的任务存储在图 10 所示的 *worklist* 结构中(第 9 行–第 16 行), *worklist* 结构一共由首、尾两部分组成.

- 首部固定占用 16 字节, 前 8 字节(*local_share_count*)用于存储 worklist 的本地共享计数器(第 10 行), 初始状态为 0, 供本机线程间的任务抢夺使用; 后 8 字节(*remote_share_count*)用于存储 worklist 的远端共享计数器(第 11 行), 初始状态为工作队列中的总任务数, 供远端节点任务迁移使用.
- 尾部采用 AoS (array of struct)结构来存储实际的工作任务(第 12 行–第 15 行), 每个工作任务由顶点 Id 和该顶点的状态组成.

```
1.  struct {
2.      uint64_t nodeId;   //计算节点的 Id
3.      uint64_t vertex_num;   //子图的总顶点数
4.      uint64_t edge_num;   //子图的总边数
5.      uint64_t local_subgraph_address;   //本地子图的存储首地址
6.      uint64_t worklist_address;   //worklist 的首地址
7.  } nodeAdrMsg;
8.
9.  struct worklist{
10.     uint64_t local_share_count;   //worklist 的本地共享计数器
11.     uint64_t remote_share_count;   //worklist 的远端共享计数器
12.       struct {
13.           vertex_id_type vertexId;   //顶点 Id vertex_id_type vertexId;
14.           vertex_data_type vertexValue;   //顶点状态
15.     } work[count];
16. }
```

图 10　任务迁移用到的结构体

由第 5.1 节可知, 本地的共享计数器可以区分 worklist 中已分配和未分配的任务. 因此, 节点可以通过获取远端计算节点的 *local_share_count* 和 *remote_share_count* 来判断远端计算节点的当前状态, 如公式(4)所示.

$$\text{节点状态}\begin{cases}\text{工作}, & \text{若} local\_share\_count < remote\_share\_count \\ \text{空闲或迁移}, & \text{若} local\_share\_count \geqslant remote\_share\_count\end{cases} \tag{4}$$

此外, 虽然 RDMA 具有高效的传输速率, 但基于 RDMA 的远端内存访问仍比基于共享内存的本地内存访问慢 20 倍. 因此, RGraph 设定了如下机制: 首先, RGraph 采用“少拿多取”的思想来保证任务的迁移性能, 每个线程单次从远端的 worklist 中获取 *migrate_block* 个任务执行迁移, *migrate_block* 默认值为 8; 其次, RGraph 设置了一个任务迁移阈值 *migrate_threshold*, 保证在剩余少量任务时, 本地线程优先执行, 即当计算节点的剩余任务小于 *migrate_threshold* 时, 该节点上的任务不在参与迁移; 最后, RGraph 中的迁移节点采用环状的拓扑结构来执行任务迁移, 以最小化迁移过程所需的 RDMA 原子操作. 算法 3 给出了任务迁移的具体步骤. RGraph 节点间任务迁移主要分为 3 步: 第 1 步, 迁移节点按序遍历集群中的其余节点(第 1 行、第 2 行), 通过 *RDMA_READ* 获取遍历节点的共享计算器, 并判断遍历节点是否为高负载节点(第 7 行–第 9 行); 第 2 步, 如果为高负载节点, 则调用 RDMA 原子操作更改高负载节点的远端共享计数器, 保证与其他迁移节点的迁移一致性(第 11 行); 第 3 步, 迁移节点端的所有线程并行调用 *RDMA_READ* 获取需要迁移的任务和任务所需的原数据, 并完成计算(第 13 行–第 17 行). 第 4 行的 while 操作保证迁移节点在遍历到高负载节点后, 会执行多轮任务迁移, 直到当前高负载节点变为空闲或迁移节点为止. 此外, 迁移节点基于迁移任务更新的顶点状态不会被立刻发送回原先的高负载节点端, 而选择暂存在自身的顶点状态数组中, 在集群的同步阶段统一处理. 同计算节点间消息传递一样, RGraph 的任务迁移模型支持线程的并行执行, 能够保证高效的任务迁移性能, 这主要受益于 RGraph 高效的通信模型.

算法 3. RGraph 的节点间任务迁移.

```
1.  For offset:1–nodeNum
2.      migrateId=(nodeId+offset)%nodeNum;   //依次遍历其余计算节点
3.      shareCount={local_share_count,remote_share_count};
4.      While true
5.          RDMA_READ(migrateId,worklist_address,shareCount);   //获取本地和远端共享计数器
```

```
6.      //① 判断是否需要迁移
7.      If (remote_share_count-local_share_count)≤migrate_threshold
8.        break;
9.      End If
10.     //② 原子的修改远端共享计数器
11.     RDMA_FETCH_AND_ADD(0,migrateId,remote_share_count,-(migrate_block×num_thread))
12.     //③ 并行获取需要迁移的任务和原数据,并计算
13.     Parallel
14.       RDMA_READ(migrateId,worklist_address,remoteWork);  //获取需要迁移的任务
15.       RDMA_READ(migrateId,local_subgraph_address,remoteOriginalData);  //获取原数据
16.       COMPUTE(remoteWork,remoteOriginalData);  //计算
17.     End Parallel
18.   End While
19. End For
```

5.2.2 迁移缓存优化

本文在对 RGraph 的任务迁移策略测试时, 发现有许多顶点多次参与到任务迁移过程中. 主要原因是: 图计算需要经过多次迭代才可以获得最终结果, 而在迭代过程中, 部分顶点会被多次压入到 worklist 中, 这部分顶点就有很大概率多次参与到任务迁移过程中. 根据这一发现, RGraph 采用迁移缓存来优化任务迁移. 具体而言, RGraph 在每个机器的本地内存中维护一块缓存, 用于存储所有线程从高负载节点获取到的原图数据. 迁移节点端的线程开始执行任务迁移时, 会优化查询迁移缓存中是否存有任务所需的原图数据: 如果有, 则跳过原数据的获取步骤(算法 3 的第 15 行), 直接开始计算. 迁移节点内的线程同样采用任务抢夺机制来保证线程间的负载均衡, 唯一不同的是, 单次任务抢夺数定义为更细的粒度. 为了保证多线程能够高效地访问迁移缓存, RGraph 采用 TBB 的 *concurrent_unordered_map* 作为原图数据的存储结构. 图 11 给出了 RGraph 的迁移缓存结构.

```
1.  struct {
2.    uint64_t insertId;  //记录顶点插入顺序
3.    uint64_t hitTime;  //顶点在缓存中的命中次数
4.  struct remoteOriginalData;  //远端获取的数据
5.  } cacheData;
6.  concurrent_unordered_map<vertex_id_type, cacheData>cache;
```

图 11 迁移缓存的结构

RGraph 还考虑到了迁移缓存的性能衰减和内存消耗问题. 首先, 哈希表虽然能够提升查询效率, 但随着缓存数据的不断增大, 迁移缓存的性能仍会受到影响; 其次, 缓存数据的增大还会造成内存的急剧消耗, 为此, RGraph 设置了缓存的最大存储数 *MAX_CACHE*, *MAX_CACHE* 以顶点为中心进行计数, 即缓存区允许存储 *MAX_CACHE* 个顶点的原图数据, 超过的部分, RGraph 根据缓存中顶点的访问频率进行删除, 优先删除访问频率较少的顶点数据. 迁移缓存的维护会在每轮迁移任务结束后执行. RGarph 的缓存替换策略如算法 4 所示.

算法 4. RGraph 的缓存替换策略.

```
1.  moreCache=cache.size(·)–MAX_CACHE;
2.  For hit:1–currentInsertId
3.    If moreCache≤0
4.      break;
5.      tempCache=moreCache;
6.    End If
```

```
7.    it=cache.begin(·)
8.    while it=cache.end(·)
9.      If (it→second.hitTime≤hit)&&(cuurrentInsertId-it→second.insertId)>4
10.       delete it;
11.       moreCache=moreCache−1;
12.     End If
13.   End while
14.   If tempCache==moreCache
15.     break;
16. End for
```

算法 4 的第 1 行首先计算缓存的远程顶点数是否超过了设置的最大存储数 *MAX_CACHE*. 第 2 行通过循环设置命中次数来优先删除最小命中次数的顶点. 第 8 行依次遍历缓存中的所有缓存顶点. 第 9 行为 RGraph 的缓存顶点删除策略: 首先, 一个顶点在被加入到缓存中的前 4 次迁移内, 不会被删除, 主要是为了避免后加入缓存的顶点总被优先删除的情形. 在满足此条件的前提下, 依次删除访问频率最小的顶点.

6 实 验

RGraph 采用 C++语言编写, 共计 7 800 行代码. 在本节中, 我们将 RGraph 与经典的分布式图处理系统进行比较, 包括 PowerGraph、Gemini. 其中, PowerGraph[10]是经典的分布式图处理系统, 基于边切的图划分使得它在处理幂律图时, 被证明有较高的性能优势; Gemini[11]凭借轻量级的系统设计和基于任务窃取的节点内负载均衡, 保证了较优的图计算性能, 被认为是当前最先进的分布式 CPU 处理系统[26](注: GraM[22]是基于 RDMA 的分布式图处理, 主要研究系统的可扩张性, 但由于 GraM 是微软的商用系统(作者邮件回复), 无法获取到 GraM 的源码).

6.1 实验配置

本文在一个具有 8 个节点的多核服务器集群上评估 RGraph. 每台服务器包括两块 Intel Xeon Silver 4214 @2.20 GHz CPU 处理器, 并具有 160 GB 内存(其中一台具有 256 GB 的内存)和一张 Mellanox ConnectX-3 56 Gb/s InfiniBand 网络接口卡, 并通过 PCIe 3.0 x8 连接一台 Mellanox 40 Gb/s 的 IB 交换机.

本文使用 6 个大图数据集对 RGraph 进行评估. 如表 1 所示, 其中, 前 5 个数据集为真实的公开可获取的社交网络数据集或 Web 数据集[7,27,28], 边规模在 1.01 亿–425 亿之间. 数据集 rmat 是本文以 R-MAT[29]工具为基础人工合成的幂律图, 图中顶点的最大出度达到了 58 797 981. Rmat 数据集主要用于测试 RGraph 在处理极度偏斜的幂律图时的性能. 表 1 还给出了数据集的其他信息, 包括平均度、最大的出度和图数据的二进制文件的大小, 图中的每条边由 3 部分组成: 源顶点-目的顶点-边值, 每个变量的基本存储单位为无符号 32 位整型, 因此每条边共占有 12 字节.

表 1 输入及其关键特性

	数据集					
	enwiki-2013	twitter-2010	uk-2007-05	weibo-2013	clueweb-12	rmat
$\|V\|$ (M)	4	41	106	73	979	1 121
$\|E\|$ (M)	101	1 468	3 739	6 432	42 575	43 388
$\|E\|/\|V\|$	24.09	35.253	35.306	88.836	43.514	38.704
max D_{out}	8 104	2 997 469	15 402	35 231	7 447	58 797 981
Edges size (GB)	1.133	16.41	41.79	71.87	475.81	520.77

本文采用 4 个经典的图算法来测试 RGraph, 包括 BFS、SSSP、PR、WCC. 其中, BFS 和 SSSP 算法源顶点的选取服从以下规则: 从每个图数据中选取 20 个顶点作为算法的源顶点集, 每个顶点必须保证具有 5 跳以上的出边邻居. PR 算法默认迭代 20 次.

6.2 RGraph的整体性能

表 2 展示了 RGraph 在 8 节点集群上与 Powergraph 和 Gemini 的性能对比. 从表中可以发现, RGraph 在 5 个真实大数据集合、1 个合成数据集下运行所有测试算法都具有明显的性能优势. 相比于 Powergraph, RGraph 平均具有 12.4 倍的加速比. 表 2 中的“X”表示计算错误, 主要原因是运行所需的内存超出计算节点拥有内存资源. 这也进一步说明了 PowerGraph 对内存资源的消耗: 相比与当前最先进的分布式 CPU 图处理系统 Gemini, RGraph 平均具有 3.7 倍的加速比, 且在运行 rmat 数据集时, RGraph 表现出了更高的加速比. 主要原因是: 相比于 Gemini, RGraph 不仅具有高效的通信模型, 还有灵活的负载均衡机制, RGraph 的负载均衡机制保证了 RGraph 在面对节点负载不均时能够实时调节, 保证计算资源的充分利用; 而 Gemini 只能通过图划分来简单地保证节点间的负载均衡. 表 2 的最后一列详细记录了不同数据和不同算法下, RGraph 相对于 Gemini 的加速比.

表 2 RGraph 的整体性能

算法	数据集	系统(s)			加速比(倍)
		Powergraph	Gemini	RGraph	
BFS	enwiki-2013	10.0	0.78	**0.43**	1.81
	twitter-2010	51.3	13.7	**3.2**	4.28
	uk-2007-05	81.8	40.5	**7.9**	5.12
	weibo-2013	150.5	51.3	**16.2**	3.16
	clueweb-12	X	136.4	**36.8**	3.7
	rmat	X	173.9	**37.0**	4.7
PR	enwiki-2013	45	20	**16.5**	1.21
	twitter-2010	261.1	93.5	**31.3**	2.89
	uk-2007-05	357.5	128.6	**40.9**	3.11
	weibo-2013	787.7	306.0	**77.2**	4.0
	clueweb-12	X	1 312.5	**370.6**	3.54
	rmat	X	1 499.2	**395.1**	3.8
SSSP	enwiki-2013	10.7	1.35	**0.63**	2.14
	twitter-2010	111.3	28.2	**7.4**	3.80
	uk-2007-05	173.3	62.7	**15.6**	4.02
	weibo-2013	391.5	118.5	**37.5**	3.16
	clueweb-12	X	290.6	**74.3**	3.91
	rmat	X	422.6	**82.1**	5.14
WCC	enwiki-2013	10.1	1.25	**0.61**	2.04
	twitter-2010	98.4	31.2	**9.8**	3.18
	uk-2007-05	76.3	34.83	**10.6**	3.28
	weibo-2013	189.3	65.7	**18.4**	3.57
	clueweb-12	X	148.3	**41.0**	3.6
	rmat	X	185.2	**50.5**	3.7

本文对系统的内存使用情况也进行了测试. 表 3 给出了部分图数据集下, 系统的内存使用情况(原图数据采用二进制存储). PowerGraph 占用内存空间最大, 主要原因来自于系统调用了大量的指令集, 如维护 GAS 抽象、顶点的局部 ID 与全局 ID 的映射等. Gemini 大约需要占用 2.3 倍的内存空间, 主要原因是: Gemini 为了维护 Push-Pull 双重更新模型, 需要分别存储 CSR 和 CSC 两种格式的图数据, 占用双倍于图文件大小的内存. RGraph 占用最少的内存空间, 大约为原图数据的 1.5 倍; 且随着图数据的增大, 占用比率会更小. 这主要源于 RGraph 的轻量级设计以及对 RDMA 内存使用的优化.

表 3 RGraph 的内存消耗 (GB)

数据集	原图大小	PowerGraph	Gemini	RGarph
uk-2007-05	41.8	489.5	117.4	70.6
weibo-2013	71.9	639.9	189.4	114.3
clueweb-12	475.8	X	870	665.5
rmat	520.8	X	902.5	718.5

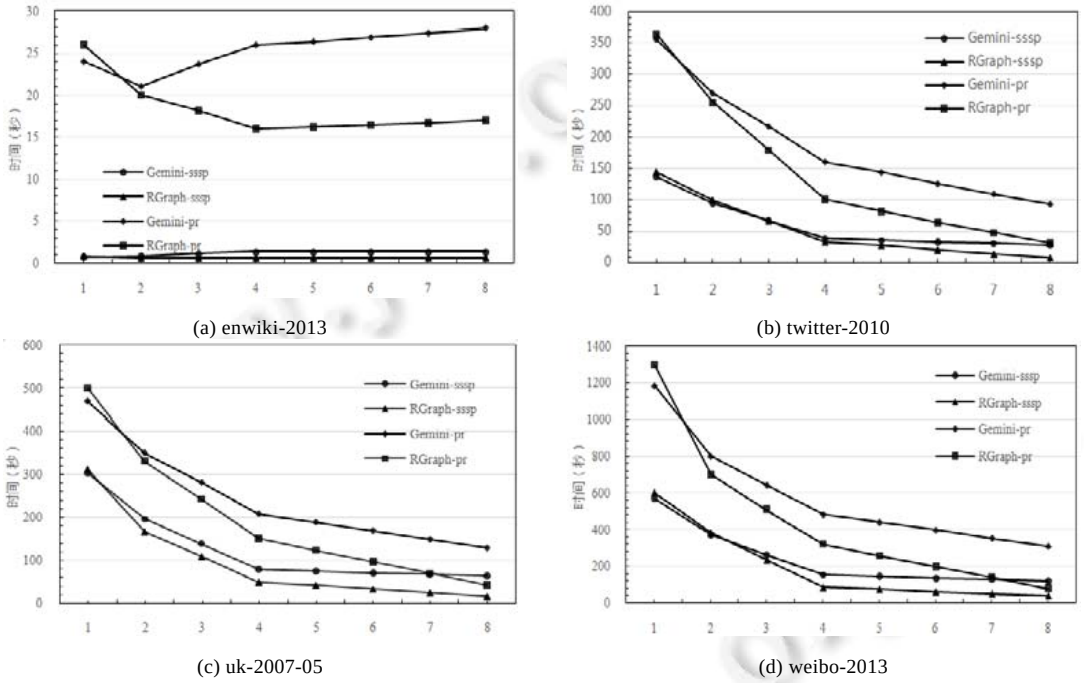
6.3 RGraph的可扩展性

本节测试 RGraph 节点间的可扩张性. 本文分别在 enweiki-2013、twitter-2010、uk-2007-05、weibo-2013

数据集下进行测试, 算法选用 SSSP 算法和 PR 算法.

如图 12 所示, 在单台机器上, Gemini 的性能会略优于 RGraph 的性能. 主要原因是: Gemini 使用了基于 NUMA 架构的优化和 Push-Pull 双重更新模型, 而 RGraph 仅仅采用了 Push 更新模型. RGraph 不采用 Gemini 中的优化, 有如下原因.

- 首先, 相比于 Gemini, RGraph 需要花费额外内存用于通信模型(RDMA 的固有特性), 因此, RGraph 放弃了需要花费双倍于图文件大小的 Push-Pull 双重更新模型, 而采用内存消耗更小的 Push 模型.
- 其次, NUMA 的优化, 使得计算节点需要以 socket 为单位处理 worklist 内部的顶点, 这会导致 RGraph 需要更加复杂的任务迁移逻辑, 如每个计算需要为每个 socket 维护一个 worklist、基于 socket 重新定义任务迁移的规则等, 这不符合 RGraph 轻量级的设计初衷. 因此, RGraph 放弃了基于 NUMA 架构的优化.
- 最后, RGraph 主要为分布式图系统设计, 从图 12 中不难发现: 随着计算节点的增加, RGraph 的性能会大幅提升, RGraph 可以从现有的选择中获取比 NUMA 和 Push-Pull 优化的更高收益.



(a) enwiki-2013 (b) twitter-2010

(c) uk-2007-05 (d) weibo-2013

图 12 RGraph 的节点间可扩展性

RGraph 的性能会随着计算顶点的增加而大幅度提升. 主要原因是:分布式图计算是通信密集型, 相比于 Gemini, RGraph 能够充分利用所有的计算资源, 避免资源的闲置, 这都归功于 RGraph 拥有更高效的通信模型; 其次, RGraph 还有动态的负载均衡机制, 不需要完全地依赖于初始的图划分.

6.4 RGraph的优化评估

本节我们将对 RGraph 各个方面的优化进行逐个分析和测试, 我们在 rmat 数据集下通过执行 SSSP 算法和 PR 算法来开展测试. 为了保证实验的准确性, 每种算法运行 3 次. 其中, SSSP 算法选取不同的源顶点进行迭代. 实验结果见表 4.

- BASE: 最基本的分布式图处理系统, 仅使用了 RGraph 的图划分、更新模型. 通信模型采用 MPI 的漏斗作为通信模型.
- +RDMA: 在 BASE 版本的基础上引入了 RGraph 的基本通信模型(第 4.1 节), 替换了之前的 MPI 漏斗

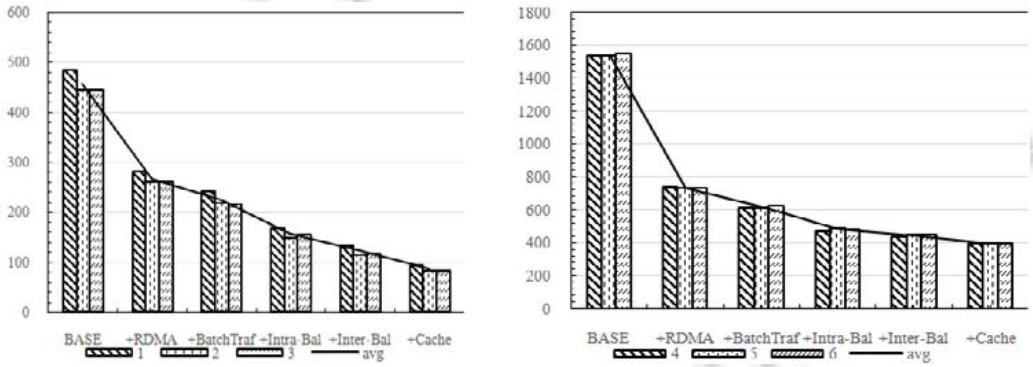
模型, 但不包括对 RGraph 通信模型的优化.

- +BatchTraf: 在前 2 步的基础上添加了消息的批传输优化(第 4.3 节).
- +Intra-Bal: 在前 3 步的基础上引入了计算节点内的负载均衡机制(第 5.1 节).
- +Inter-Bal: 在前 4 步的基础上引入了节点间的负载均衡机制, 但不包括迁移缓存优化(第 5.2.1 节).
- +Cache: 在前 5 步的基础上引入了迁移缓存优化(第 5.2.2 节).

表 4 RGraph 的优化 (s)

算法	序号	BASE	+RDMA	+BatchTraf	+Intra-Bal	+Inter-Bal	+Cache
SSSP	1	484.4.	242.2	201.9	167.6	132.9	93.3
	2	443.3	230.4	193.8	148.1	113.9	82.5
	3	444.7	230.5	192.5	153.5	117.0	82.0
PR	1	1 540.0	739.9	616.6	474.3	439.1	395.1
	2	1 536.7	731.8	615.2	492.4	451.4	396.4
	3	1 547.9	737.1	624.1	485.6	449.6	397.6

总体而言, 最终的 RGraph 系统相比于 BASE 版本能获得 4.49 倍的性能提升. 对于不同的算法, 每个策略对性能的影响也不相同. 对于 SSSP 算法, RGraph 能够从+RDMA、+Inter-Bal 和+Cache 获得较高的性能收益, 如图 13(a)所示. 主要原因是: 首先, RGraph 的通信机制能够有效地提升通信密集型的分布式图处理系统的性能;其次, RGraph 的负载均衡策略能够有效地服务于 SSSP 在迭代过程中导致的节点间负载不均. 对于 PR 算法, RGraph 更多的收益来源于+RDMA 和+BatchTraf, 如图 13(b)所示. 这是因为 PR 算法在迭代过程中几乎所有的顶点状态都需要参与传输, 系统的通信量较高, 因此基于 RDMA 的通信模型可以充分发挥通信优势. 上文提到的所有优化都是保证 RGraph 高性能的基础, 它们使得系统在接收任何输入时, 如极度偏斜的幂律图、通信密集型应用、计算密集型应用等, 都能有稳定的性能收益.



(a) SSSP (b) PR

图 13 RGraph 的节点间可扩展性

6.4.1 通信模型对比

本节将从 CPU 利用率和不同数据块下的通信延迟两个方面, 对传统 MPI 漏斗模型和本文提出的基于 RDMA 的异构并行模型进行对比.

如图 14 所示, 我们每秒记录一次单个计算节点所有工作线程的 CPU 平均利用率. 可以发现, RGraph 的异构并行模型平均具有 81%的 CPU 平均利用率, 而传统 MPI 漏斗模型的 CPU 平均利用率为 69%. 异构并行模型拥有更高的 CPU 利用率主要原因是: MPI 的漏斗模型在 Gather 阶段需要所有工作线程串行执行, 导致多数线程出现排队等待状态, 此时 CPU 利用率会大幅度降低, 如图 14 中的 9 s 和 21 s 处. RDMA 的异构并行模型会在每个线程周期处理各自 QP 连接的 poll 函数时, CPU 利用率会少许较低, 如图 14 的 12 s 和 24 s 处.

图 15 是 RGraph 基于 RDMA 的异构并行模型和传统基于 MPI 的漏斗模型在不同数据块下通信延迟的对比, 其中, x 轴表示不同的传输数据块, y 轴表示通信延迟. 对于小于 4 KB 的数据块, RGraph 的异构并行模型平

均具有 6 μs 的通信延迟, 而 MPI 漏斗模型平均具有 20 μs 的通信延迟. 数据块在大于 64 KB 后, 两种通信模型的延迟差距变大, 如: 数据块大小为 256 KB 时, 异构并行模型的通信延迟为 48 μs, 而 MPI 的漏斗模型延迟达到了 320 μs. 因此, 对于 RGraph 的通信模型, 我们定义最大的传输单元为 4 KB.

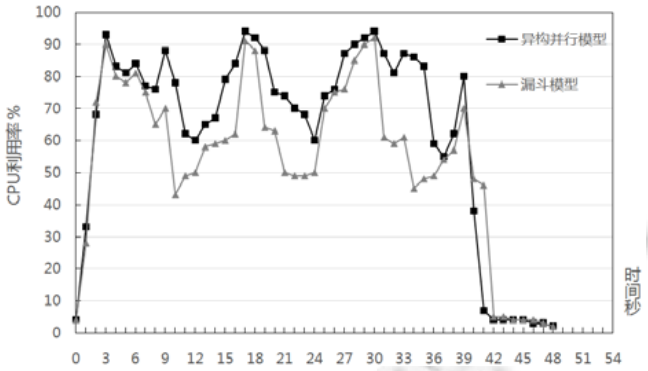


图 14 两种通信模型的 CPU 利用率

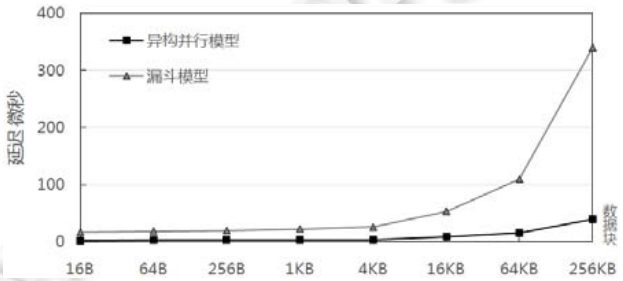


图 15 两种通信模型的延迟对比

6.4.2 节点内负载均衡

本节我们详细对本文提出的任务抢夺方式和传统的任务窃取两种不用节点内负载均衡方式进行了性能对比, 其次还分析在不同任务数下, 对任务抢夺方式的性能影响.

如表 5 所示, 本文在两个不同的数据集下, 利用 SSSP 和 PR 算法对任务抢夺机制和任务窃取机制进行了性能测评. 从实验数据可以发现: 两种负载均衡机制具有相似的性能, 任务窃取会有少许的性能的优势, 但几乎可以忽略. RGarph 不仅考虑了计算节点内的负载均衡, 还考虑了计算节点间的负载均衡, 且节点内的负载均衡是节点间负载均衡的基础, 而传统的任务窃取并不能适配计算节点间的负载均衡. 为此, RGraph 选取了任务抢夺作为基本的负载均衡方式.

表 5 任务抢夺和任务窃取性能对比 (s)

算法	数据集	+BatchTraf	+Intra-Bal	
			任务抢夺机制	任务窃取机制
SSSP	weibo-2013	80.1	61.8	62.9
	rmat	201.9	158.5	155.7
PR	weibo-2013	135.7	114.5	113.2
	rmat	615.8	492.4	491.6

表 6 为不同单次任务抢夺数下(64, 128, 256)对负载均衡性能的影响, 包括节点内的负载均衡(+Intra-Bal)和节点间的负载均衡(+Inter-Bal), 括号内为性能的提升比. 从实验数据可以发现: 单次任务抢夺数定义为 64 可以获得最高的性能提升; 单次任务抢夺数定义为 256 可以获得的性能提升最小, 甚至会使 RGraph 的节点间任务迁移机制失效. 如: 在 PR 算法下, 256 的单次任务抢夺数使得节点间的任务迁移机制仅获得 3%的性能提升. 主要原因是: 单次任务抢夺数太大, 任务抢夺机制使得单个线程将大量任务占位私有, 其余空闲线程或

计算节点无法获取未完成的任务, 造成计算资源的浪费.

表 6 任务抢夺和任务窃取性能对比 (s)

算法	任务数	+BatchTraf	+Intra-Bal	+Inter-Bal
SSSP	**64**	198.8	168.6 (15%)	132.9 (21%)
	128	189.8	179.5 (9%)	157.7 (12%)
	256	193.8	175.6 (11%)	168.1 (4%)
PR	**64**	618.4	484.4 (21%)	419.1 (13%)
	128	617.2	540.4 (12%)	500.4 (7%)
	256	614.4	550.5 (11%)	531.5 (3%)

7 相关工作

近年来, 随着社交网络等新型应用的兴起和云计算等新技术的快速发展, 图数据的重要性和规模的都在不断增大, 如何并行地处理图数据, 已经成为学术界和商业界重要的研究领域. 在此背景下, 出现了许多图处理系统. 其中, 有的系统[12−15,30−32]侧重基于多核处理器的单计算节点优化, 另外的系统[8−11,16,33,34]侧重于研究分布式处理. TurboGraph++[34]和 Chao[35]是基于分布式的大图数据处理系统, 其中, TurboGraph 提出了一种 3-LPO 的处理策略, 使得 CPU 计算任务、磁盘 I/O 任务、网络 I/O 任务可以重叠, 以实现系统的有效性和可扩展性; 而 Chao 可以在 32 台机器上用 2 天时间处理完 1 万亿条边的图数据, 这与它使用的二级存储有关. 虽然 RGraph、TurboGraph++和 Chao 同样为分布式图处理系统, 但 RGraph 是基于内存的分布式图处理系统, 而 TurboGraph++和 Chao 是基于多级存储的分布式图处理系统.

Powergraph[10]和 PowerLyra[16]是经典的基于内存的分布式图处理系统, Powergraph 提出了基于点切的图划分方式, 保证了集群在处理幂律图时的均衡性. Powergraph 还利用 Master-Mirror 来表示顶点的切割和复制, 来抽象计算节点的消息传递. RGraph 借鉴了这种通信抽象, 基于 Master-Mirror 来搭建自己的更新模型. 不同于 Powergraph 的是: RGraph 不完全依赖于图分割来保证计算节点的均衡, RGraph 拥有动态的负载均衡机制.

GraM[22]设计了基于 RDMA 的通信协议来作为系统的通信模型, 为的是充分利用 RDMA 单边内存原语来重叠计算时间和通信时间. RGraph 采用了相同的理念, 但 RGraph 还同时考虑了分布式图处理系统的通信特点, 针对 RDMA 进行了符合图计算语义的优化. 由于 GraM 为微软商用系统, 所以我们无法具体了解 GraM 的内部处理逻辑和优化. Gemini[11]被认为是当前最先进的分布式 CPU 图计算系统, 它实现了基于块的图数据划分、Push-Pull 双重更新模型、节点内基于任务窃取的负载均衡以及基于 NUMA 的数据访问优化, 这些都使得 Gemini 有较优的系统性能. 图划分方面, RGraph 借鉴了 Gemini 的基于块的轻量级图划分方式, 但采用了点边综合考虑的方式进行优化, 提升了划分性能; 负载方面, 不同于 Gemini, RGraph 采用了任务抢夺的方式来保证节点内的负载均衡. 此外, RGraph 还有节点间基于任务迁移的负载均衡机制, 这使得 RGraph 不会过度依赖于图划分来保证负载均衡, 能够在复杂的图数据下获得稳定的性能收益.

8 总 结

本文通过对经典的分布式图处理系统的深入研究, 总结归纳出了影响它们性能的 3 大挑战. 针对这些挑战, 本文提出了 RGraph, 一个基于 RDMA 的高效分布式大图数据处理系统. RGraph 旨在通过充分利用 RDMA 的优势来提升图处理系统的性能. 具体而言, 图划分方面, RGraph 提出了新的基于块的图划分策略, RGraph 的图划分策略从顶点和边的混合角度出发来综合考虑分布式图计算的均衡问题, 不仅能够避免破坏原始图数据的数据局部性, 保证顶点的高效访问, 还能从图划分角度提高节点间的负载均衡; 通信方面, RGraph 为图计算编写了基于 RDMA 的高效通信模型, RGraph 的通信模型采用 *WRITE* 操作实现节点间的消息传递, 采用 *READ* 操作去实现远程无感知的任务调度. 为了进一步优化通信模型, RGraph 提出了基于可复用结构的 RDMA 内存优化和基于消息批传输的通信优化. RGraph 还提供了与 MPI 相似的调用接口, 隐藏了复杂的 RDMA 通信逻辑, 使得未接触过 RDMA 的用户也可以轻松调用 RDMA 来实现高效的通信性能. 负载方面,

RGraph 引入了动态的负载均衡技术, 包括基于任务抢夺的计算节点内线程间的负载均衡和基于任务迁移的计算节点间的负载均衡, 来共同保证图计算的负载均衡. 对于计算节点的负载均衡, RGraph 还通过设计迁移缓存来避免频繁的远程内存读取带来的性能损失, 保证任务迁移的高效性. API 方面, RGraph 为用户提供了轻量级的程序接口, 接口隐藏了复杂的分布式处理逻辑, 用户只需指定合适的数据类型, 并根据想要实现的算法逻辑编写少量的程序代码, 就可以轻松地在 RGraph 高效地运行自定义的大图算法.

最后, 与当前最先进的基于 CPU 的分布式图处理系统相比, RGraph 平均拥有 3.7 倍的性能提升, 且在极度偏斜的幂律图上仍能保证稳定的性能优势.

References:

[1] 2021. https://en.wikipedia.org/wiki/Big_data

[2] Yuan Y, Lian X, Wang G, *et al*. Constrained shortest path query in a large time-dependent graph. Proc. of the VLDB Endowment, 2019, 12(10): 1058−1070.

[3] Qiu X, Cen W, Qian Z, Peng Y, Zhang Y, Lin X, Zhou J. Real-time constrained cycle detection in large dynamic graphs. Proc. of the VLDB Endowment, 2018, 11(12): 1876−1888.

[4] Bronson N, Amsden Z, Cabrera G, *et al*. Tao: Facebook's distributed data store for the social graph. In: Proc. of the USENIX Annual Technical Conf. Association for Computing Machinery, 2013. 49−60.

[5] Shi J, Yao Y, Chen R, Chen H, Li F. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2016. 317−332.

[6] Zhang Y, Chen R, Chen H. Sub-millisecond stateful stream querying over fast-evolving linked data. In: Proc. of the 26th Symp. on Operating Systems Principles. Association for Computing Machinery, 2017. 614−630.

[7] 2021. http://law.di.unimi.it/webdata/clueweb12/

[8] Malewicz G, Austern MH, Bik AJC, *et al*. Pregel: A system for large-scale graph processing. In: Proc. of the SIGMOD. Association for Computing Machinery, 2010. 135−146.

[9] Xin RS, Gonzalez JE, Franklin MJ, *et al*. GraphX: A resilient distributed graph system on spark. In: Proc. of the Graph Data Management Experiences and Systems. Association for Computing Machinery, 2013. 1−6.

[10] Gonzalez JE, Low Y, Gu H, *et al*. PowerGraph: Distributed graph-parallel computation on natural graphs. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2012. 17−30.

[11] Zhu X, Chen W, Zheng W, *et al*. Gemini: A computation-centric distributed graph processing system. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2016. 301−316.

[12] Nguyen D, Lenharth A, Pingali K. A lightweight infrastructure for graph analytics. In: Proc. of the Int'l Parallel and Distributed Processing Symp. Institute of Electrical and Electronics Engineering, 2013. 456−471.

[13] Shun J, Blelloch GE. Ligra: A lightweight graph processing framework for shared memory. In: Proc. of the Principles and Practice of Parallel Programming. Association for Computing Machinery, 2013. 135−146.

[14] Sundaram N, Satish NR, Patwary MMA, *et al*. GraphMat: High performance graph analytics made productive. Proc. of the VLDB Endowment, 2015, 8(11): 1214−1225.

[15] Zhang K, Chen R, Chen H. NUMA-aware graph-structured analytics. In: Proc. of the Principles and Practice of Parallel Programming. Association for Computing Machinery, 2015. 183−193.

[16] Chen R, Shi JX, Chen YZ, *et al*. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. ACM Trans. on Parallel Computing, 2018, 5(3): Article No.13.

[17] Valiant LG, *et al*. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8): 103−111.

[18] Low YC, Gonzalez J, Kyrola A, *et al*. Distributed GraphLab: A framework for machine learning in the cloud. Proc. of the VLDB Endowment, 2012, 5(8): 716−727.

[19] Stanton I, Kliot G. Streaming graph partitioning for large distributed graphs. In: Proc. of the Knowledge Discovery and Data Mining. Association for Computing Machinery, 2012. 1222−1230.

[20] Bourse F, Lelarge M, Vojnovic M. Balanced graph edge partition. In: Proc. of the Knowledge Discovery and Data Mining. Association for Computing Machinery, 2014. 1456−1465.

[21] Pingali K, Nguyen D, Kulkarni M, *et al*. The tao of parallelism in algorithms. In: Proc. of the Programming Language Design and Implementation. Association for Computing Machinery, 2011. 12−25.

[22] Wu M, Yang F, Xue J, *et al*. Gram: Scaling graph computation to the trillions. In: Proc. of the Symp. on Cloud Computing. Association for Computing Machinery, 2015. 408−421.

[23] Dang HV, Dathathri R, Gill G, *et al*. A lightweight communication runtime for distributed graph analytics. In: Proc. of the Int'l Parallel and Distributed Processing Symp. Institute of Electrical and Electronics Engineering, 2018. 980−989.

[24] Dang HV, Snir M, Gropp W. Towards millions of communicating threads. In: Proc. of the EuroMPI. Association for Computing Machinery, 2016. 1−14.

[25] Vaidyanathan K, Kalamkar DD, Pamnany K, *et al*. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In: Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, 2015. 1−12.

[26] Dathathri R, Gill G, Hoang L, *et al*. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytic. In: Proc. of the Programming Language Design and Implementation. Association for Computing Machinery, 2018. 752−768.

[27] Han W, Zhu X, Zhu Z, *et al*. Weibo, and a tale of two worlds. In: Proc. of the Advances in Social Networks Analysis and Mining. Association for Computing Machinery, 2015. 121−128.

[28] Boldi P, Rosa M, Santini M, *et al*. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Proc. of the World Wide Web. Association for Computing Machinery, 2011. 587−596.

[29] Chakrabarti D, Zhan Y, Faloutsos C. R-MAT: A recursive model for graph mining. In: Proc. of the SIAM Int'l Conf. on Data Mining. Society for Industrial and Applied Mathematics, 2004. 442−446.

[30] Kyrola A, Blelloch G, Guestrin C. GraphChi: Large-scale graph computation on just a PC. In: Proc. of the Operating Systems Design and Implementation. USENIX Association, 2012. 31−46.

[31] Prabhakaran V, Wu M, Weng X, *et al*. Managing large graphs on multi-cores with graph awareness. In: Proc. of the Annual Technical Conf. USENIX Association, 2012. 41−52.

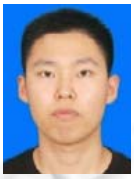
[32] Roy A, Mihailovic I, Zwaenepoel W. X-Stream: Edge-centric graph processing using streaming partitions. In: Proc. of the Symp. on Operating Systems Principles. USENIX Association, 472−488.

[33] Murray DG, McSherry F, Isaacs R, *et al*. Naiad: A timely dataflow system. In: Proc. of the Symp. on Operating Systems Principles. USENIX Association, 2013. 439−455.

[34] Ko S, Han WS. TurboGraph++: A scalable and fast graph analytics system. In: Proc. of the SIGMOD. Association for Computing Machinery, 2018. 395−410.

[35] Roy A, Bindschaedler L, Malicevic J, *et al*. Chaos: Scale-out graph processing from secondary storage. In: Proc. of the Symp. on Operating Systems Principles. Association for Computing Machinery, 2015. 410−424.

崔鹏杰(1993－), 男, 博士生, 主要研究领域为新型硬件, 数据库理论与系统.



张灿(1997－), 男, 硕士生, 主要研究领域为新型硬件, 数据库理论与系统.

袁野(1981－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据管理, 数据库理论与系统.

王国仁(1966－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为不确定数据管理, 数据密集型计算, 可视媒体数据分析管理, 非结构化数据管理, 分布式查询处理与优化, 生物信息学.

李岑浩(1996－), 男, 硕士生, CCF 学生会员, 主要研究领域为新型硬件, 数据库理论与系统.