硕 士 学 位 论 文

大规模图增量迭代处理技术的研究与实现

导　师：于戈　教授

研究生：王志刚

东北大學

二〇一三年六月

分类号_____密级_____

UDC _____

学 位 论 文

大规模图增量迭代处理技术的研究与实现

作 者 姓 名 ： 王志刚

指 导 教 师 ： 于戈 教授

东北大学信息科学与工程学院

申请学位级别： 硕士 学 科 类 别 ： 工学

学科专业名称： 计算机软件与理论

论文提交日期： 2013 年 6 月 论 文 答 辩 日 期 ： 2013 年 6 月

学位授予日期： 2013 年 7 月 答辩委员会主席： 王大玲

评 阅 人 ： 谷峪，孙焕良

东 北 大 学

2013 年 6 月

A Thesis in Computer Software and Theory

Research and Implementation on Incremental and Iterative Processing Techniques for Large-Scale Graphs

by Wang Zhigang

Supervisor: Professor Yu Ge

Northeastern University

June 2013

独创性声明

本人声明，所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人已经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：

日　　期：

学位论文版权使用授权书

本学位论文作者和指导教师完全了解东北大学有关保留、使用学位论文的规定：即学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人同意东北大学可以将学位论文的全部或部分内容编入有关数据库进行检索、交流。

作者和导师同意网上交流的时间为作者获得学位后：

半年 □　　一年□　　一年半□　　两年□

学位论文作者签名：　　导师签名：

签字日期：　　签字日期：

大规模图增量迭代处理技术的研究与实现

摘 要

社交网络、生物信息网络和信息技术的快速发展，使图论及其相关算法的应用日益广泛。其中，利用云计算环境开发大规模图的增量迭代处理平台，已经成为当前学术界和工业界研究的热点。但是针对数据划分、磁盘管理和网络通信等方面的优化工作，相对较少，而这三个方面正是影响大图处理平台运行效率的至关重要的因素。由于大图的增量迭代处理，具有数据耦合性强、迭代频率高、数据访问局部性差等特点，因此对于上述三个方面的优化处理是一个十分具有挑战性的课题，这也是本文的主要工作内容。

在数据划分方面，本文首先分析了 DFS 生成图和 BFS 生成图的局部性，然后提出了连续划分策略。相比于现有的随机 Hash 划分，连续划分策略能够保留输入图的原始局部性，避免数据划分阶段的全局洗牌操作，均衡计算负载。同时，为提供迭代过程的消息路由寻址服务，本文提出了基于 Hadoop 的图顶点连续编码方案和基于 DHT 的 Hybrid-*MT* 连续编码方案，可以将图顶点字符串转换为数字连续编号。

在磁盘管理方面，通过分析增量迭代过程，本文提出了图状态转换模型，并据此设计了基于列存储模型的静态 Hash 索引和基于 Markov 模型的动态可调 Hash 索引。前者可以提高消息数据与本地图数据之间连接操作的局部性，避免随机磁盘存取。后者在前者基础上，根据迭代状态的转换和 Markov 代价收益评估模型，可以动态调整 Hash 桶的分割粒度，最小化磁盘存取开销。

在网络通信方面，通过扩展 BSP 模型（Bulk Synchronous Parallel），本文提出了基于 EBSP 模型的 Hybrid 迭代机制，即图数据同步处理，而消息数据跨步处理。特别的，消息跨步剪枝（ASMP）和跨步合并（ASMC）是两种跨步处理方法，可以分别剪枝消息规模和加速消息传播速度。此外，在连续划分基础上，本文提出了 VCCP（Vertex-Cut based on the Continuous Partitioning）方法，利用 BFS 生成图的原始局部性，VCCP 通过较低的预处理开销，能够显著改善总体运行效率。

最后，本文设计了 DiterGraph 原型系统，通过大量实验验证了数据划分、磁盘管理和消息通信的性能。与现有系统对比，对于单源最短路径算法，DiterGraph 的运行效率是 Giraph 的 2 倍，与 Hadoop 和 Hama 相比，最高可达 21 倍和 43 倍；对于 PageRank 算法，采用 VCCP 方法后，DiterGraph 的运行效率最高可达 Hadoop 的 20 倍。

关键词：大规模图；增量迭代处理；数据划分；磁盘索引；网络通信

Research and Implementation on Incremental and Iterative Processing Techniques for Large-Scale Graphs

Abstract

With the rapid development of social networks, bioinformatics networks and information technologies, the graph theory and its algorithms have attracted wide attention. For academia and industry, it has been a hot topic to develop platforms for incrementally and iteratively processing large-scale graphs in cloud computing environments. Nevertheless, few existing methods are designed to optimize data partitioning, disk buffer management and network communication, which are the three key issues for improving performance. The incremental and iterative process of large-scale graphs has three features: strong coupling of graph data, highly iterative frequency and poor locality of data-access. Therefore, it is a nontrivial task for optimizing the three key issues, which is the main work of this thesis.

Data partitioning. We analyze the locality of raw graphs generated by the depth-first search (DFS) algorithm and the breadth-first search (BFS) algorithm, and then propose the continuous partition (CP) strategy. Compared with the random hash partition (RHP) method, the CP strategy can preserve the locality of raw graphs, avoid shuffling all graph data by networks and balance the computation load. Meanwhile, to provide the addressing service for network communication during iterations, we design two encoding methods: a Hadoop-based encoding mechanism and a DHT-based Hybrid-*MT* mechanism, which can convert the string vertex identifiers into the consecutively numbered identifiers.

Disk buffer management. By analyzing the incremental and iterative process, we propose a state-transition model. Then a static hash index based on the oriented-column storage model and a tunable hash index based on the Markov model have been designed to optimize disk I/O costs. The static hash index can improve the locality of matching message data with local graph data and avoid the random data-access. Further, based on the state-transition model and the Markov cost-benefit analysis model, our tunable hash index can adjust the bucketing granularity dynamically to minimize I/O costs.

Network communication. We propose the EBSP model by extending the traditional Bulk Synchronous Parallel (BSP) model. Then we design a hybrid iteration mechanism, which means graph data must be processed synchronously but message data can be handled across two consecutive iterations. Especially, the across-step message pruning (ASMP) method and the across-step message combination (ASMC) method are two optimization strategies for message processing, which can reduce the message scale and accelerate the spread of messages respectively. Furthermore, this thesis designs a VCCP (Vertex-Cut based on the CP strategy) method by integrating the Vertex-Cute technology with the CP strategy. Benefiting from preserving the raw locality of BFS-generated graphs, the VCCP method can improve the overall performance greatly with the low overhead for preprocessing.

At last, we have designed and implemented a prototype system, namely DiterGraph, to validate the effect of our mechanisms by extensive experiments, including data partitioning, hash index, ASMP and VCCP. For the single source shortest path computation, the performance of DiterGraph compared to Giraph is roughly a factor of 2. Compared with Hadoop and Hama, it is up to 21 and 43 respectively. For the PageRank computation, the overall performance of DiterGraph is 20 times faster than that of Hadoop.

Key words： large-scale graph; incremental and iterative process; data partitioning; disk and index; network communication

目 录

第1章 引言

1.1 研究背景

1.1.1 云计算与大数据

互联网技术的快速发展和各种电子设备的日益普及，使电子数据的规模呈现爆炸式增长，“Big Data”已经成为学术界和工业界的一个热点话题。大数据主要存在于四个领域[1]：科学计算，如基因图谱分析和大脑拓扑分析，人类基因图谱已经包含了约7000PB的数据；金融交易，如电商交易数据，淘宝每天产生约50TB数据；流数据，如物联网领域的实时传感器数据，伦敦的城市交通摄像头每天获取约7TB图像；互联网数据，如搜索引擎、社交网络、视频网站等，Google每天产生24TB数据，FaceBook每天需要分析逾500TB数据，Youtube用户平均每日上传52万分钟的视频。因此，大数据已经深入到社会生活的各个方面。

大数据主要有4个特性，即4V理论[2]：数量（Volume）、多样性（Variety）、速度（Velocity）和真实性（Veracity）。如何高效的组织管理大数据并进行快速有效的分析处理，是大数据领域面临的主要挑战。而云计算技术的出现，为大数据处理提供了有力的支撑[3]。云计算是网格计算、分布式计算、并行计算、效用计算、网络存储和虚拟化等先进计算机技术与网络技术发展融合的产物。云计算技术通过网络整合大量廉价的商务计算机的存储能力和计算资源，对外提供存储与计算服务。云计算技术对用户屏蔽了底层资源的异构性，使用户能够透明的动态申请、配置自己所需的计算资源。云计算的服务是可计量化的，即用户根据自己的实际使用支付费用，是一种即付即用的服务模式。

云计算服务，主要包括：IaaS，基础设施即服务，如Amazon的EC2，用户可以租用服务资源，按照自己的意志组织存储数据并完成计算处理；PaaS，平台即服务，如微软的Azure和Apache的Hadoop等，用户可以利用服务平台开发特定应用处理程序；SaaS，软件即服务，用户直接租用软件服务，如云笔记等。本文的主要工作，就是提高云计算环境下大图处理的效率。

1.1.2 大规模图的增量迭代计算

图是计算机科学中最常用的一类抽象数据结构，特别适合表达现实世界中各种复杂的关系。许多重要应用都需要用图结构表示，传统应用如最优运输路线的确定、疾病爆发路径的预测、科技文献的引用关系等；新兴应用如社交网络分析、语义Web分析、生物信息网络分析等，与图相关的处理和应用几乎无所不在[3]。但由于图计算具有高度复

杂性，极大地限制了图的应用。例如，设N为顶点个数，最短路径计算算法的复杂性是$O(N^2)$，当N为百万量级时，复杂性达到10^{12}次，若使用1台运算速度约为1万MIPS次的普通PC机，设每次图计算需要10条指令，则以上算法需要1000秒的时间；如果N达到10亿量级，则算法执行需要10^9秒=10亿秒=31.7年。即使同时使用1万台机器并行计算，也至少需要27小时。

随着信息化时代的到来，各种应用和信息以爆炸模式增长，导致图的规模日益增大，涉及数十亿个顶点和上万亿条边的应用并不罕见。据CNNIC统计，2010年中国网页规模达到600亿个，而基于互联网的社交网络也后来者居上，如全球最大的社交网络Facebook，已有约7亿用户[3]。真实世界中这些实体数量的大规模增长，导致相应图模型的数据规模迅速增长。以搜索引擎中常用的PageRank计算[4]为例，网页用图顶点表示，网页之间的链接关系用有向边表示，设每个顶点及每条边的存储空间占100字节，则按邻接表形式存储10亿个图顶点和100亿条边，那么整个图的存储空间将超过1TB。如果图的顶点为不定长的字符串并且顶点和边具有大量的属性信息，存储开销将进一步增大。

如此大规模的图，其存储和计算处理过程中的时空开销，远远超出了传统集中式图数据管理方法的承受能力。因此，利用云计算完成大图处理，是大数据处理的一个重要的研究分支。

复杂的图算法通常是采用递归调用方式对整个图进行多次迭代计算，典型算法如单源最短路径计算、PageRank计算和连通域计算等。迭代算法通过多次循环处理数据集，使整体收敛至稳定状态，而循环过程中的中间计算结果均是不准确的。Yanfeng Zhang等人详细分析了迭代算法的特点并提出了分布式累加迭代计算概念[5]。所谓累加迭代，即满足如下条件的迭代计算模型：

$$\begin{cases} v_j^k = v_j^{k-1} \oplus \Delta v_j^k \\ \Delta v_j^{k+1} = \sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k) \end{cases}$$

其中，k为迭代步数，v_j^k表示第j个任务上的计算单元v在第k步的值，$\oplus$是用户自定义逻辑运算，必须满足交换律、结合律，且对数值0存在恒等性，即$x \oplus 0 = x$。而$g_{\{i,j\}}(\Delta v_i^k)$则表示在第$k$步，第$i$个任务发送给第$j$个任务上的计算单元$v$的累加值。因此，所谓累加迭代计算模型，主要包括两个核心计算过程：（1）第j个任务上的计算单元v的值，是利用该单元在第$k-1$步的值与累加和（即Δv_j^k）进行$\oplus$操作得到；（2）而第$k+1$步的累加和，是通过对所有单元在第k步发送给v的累加值进行$\oplus$操作，累加得到。Yanfeng Zhang等人指出，累加迭代可以被异步执行，并据此设计了异步累加迭代系统Maiter[5]，支持大数据的迭代计算。

累加迭代中的Δv_j^k依赖于所有计算单元的累加值Δv_i^k，$i = 1, 2, ..., n$，该模型适用于多种应用，如单源最短路径计算，PageRank计算，线程方程的Jacobi迭代解法等。其中，前两者属于图算法。此外，Stephan Ewen等人将分布式迭代计算分为两大类[6]：“Bulk Iteration”和“Incremental Iteration”。对于前者，第*k*步迭代计算需要依赖第*k-1*步的整体计算单元的计算结果，如k-means算法，这类算法显然不满足累加迭代约束[5]；对于后者，则仅需要依赖于第*k-1*步的部分单元的计算结果，即稀疏计算依赖。

图算法一般根据图的拓扑结构完成计算处理，因此，顶点*v*的在第*k*步的计算处理，仅依赖于其入度顶点在第*k-1*步的计算结果，如上述的单源最短路径计算、PageRank计算等，既满足稀疏计算依赖，也满足累加迭代的一般性约束，我们称这类算法为增量迭代图算法。图的增量迭代算法，即累加迭代中的Δv_j^k仅依赖于顶点*v*的入度顶点的累加值。一般的，我们称累加值为消息。在增量迭代过程中，顶点*v*收到部分入度顶点的消息，即可以启动本地计算处理而不会影响最终迭代计算结果的准确性。

1.2 研究意义

1.2.1 大图增量迭代计算的特点与挑战

根据1.1.2小节的分析，现有大图的规模通常可以达到几亿甚至几百亿顶点，存储开销达到TB乃至PB级别，而且图的规模（包括顶点、边，以及对应的属性信息）仍在快速增长。利用云计算的分布式架构处理大图，已经成为当前的研究热点[3]。然而，目前著名的图处理系统，如Pregel[7]、Giraph[8]、GPS[9]和PowerGraph[10]等，均假设全部数据驻留集群的分布式内存中。但是，一个给定集群的内存总容量是有限的，无法满足数据规模日益增长的大图处理需求。而扩展集群计算节点的方法，也受到理论和经济因素的制约，可操作性差。从理论方面分析，目前主流的云计算平台，如Hadoop[11]和Pregel等，均是Master-Slave结构，当集群中的Slave节点规模持续增大（4000台左右），Master将成为性能瓶颈[11]。从经济角度分析，虽然云计算集群依赖于廉价的计算机，但是集群的运维成本，尤其是电力成本，不可忽视。大型数据中心每月的电力开销占总运维开销的42%，一台服务器三年的电力消耗将超过其本身的购买成本[12]。根据美国环境保护署的研究报告，2006年全美服务器和数据中心的电力消耗为61亿千瓦时，费用高达45亿美元，约占美国总电力消耗的1.5%，到2011年，将达到3%左右[13]。此外，如果数据全部驻留内存，则同时运行的任务数目将受内存容量的限制，影响集群在单位时间内的吞吐率。综合考虑大图规模的持续增长、集群规模的有限性和吞吐率等因素，采用分布式磁盘处理方式，将是一个经济可靠的解决方案。

对于大图的增量迭代计算，发送给同一个目的图顶点的增量值没有“关联完整性”

约束，即不必收到所有入度顶点的增量值，就可以启动目的图顶点的本地计算，而不会影响迭代收敛时的最终计算结果的正确性。因此，分布式大图增量迭代计算具有如下特点：

（1）可合并性：发往同一个目的图顶点的消息，可以在发送端和接收端及时合并，以降低网络通信开销，节省存储开销；

（2）可异步性：本质上不需要全局同步来保证算法执行结果的正确性，即各图顶点的计算可以异步执行，特别的，每收到一条消息即可启动本地计算；

（3）状态可变性：迭代过程中，图顶点的值会逐渐收敛，达到稳定状态，稳态图顶点无消息收发，不需要在后续迭代中被处理。

而基于分布式磁盘架构的大图迭代处理具有如下挑战：

（1）数据访问频率高：图算法一般按照出度边访问图顶点，因此在一次迭代中，图顶点数据被访问的频数的上限为O(|*E*|)，其中*E*为图的出度边集合，如果是有向完全图，则|*E*| = |*V*| * (|*V*| - 1)，则对于拥有10亿顶点规模的大图，数据访问频率可达10^{18}；

（2）局部性差：按照出度边发送的增量值（即消息）到达目的图顶点时具有无序性，故目的图顶点进行的增量求和以及增量更新操作，均需要依赖于连接查询操作，局部性差；

（3）消息通信规模大：网络通信一直是影响分布式计算效率和可扩展性的重要制约因素，由于图的耦合性较强，其分布式计算过程必然导致大量的消息通信，用于发送增量值；

（4）磁盘存取开销大：如果数据驻留磁盘，前三个特性，会导致巨大的磁盘存取开销；

（5）迭代频率高：典型的，如PageRank算法，通常需要迭代几十次甚至上百次才会收敛，对于单源最短路径，迭代步数等于图的直径，高频迭代，会累计放大（1）-（4）的效应，严重影响总体执行效率。

针对上述问题，本文将利用大规模图的增量迭代的特点，研究大图的分布式划分算法、磁盘索引技术和消息通信的优化策略，以提高大规模图增量迭代的处理效率。

1.2.2 国内外研究现状

近些年来，云计算以其在大规模数据处理方面的诸多优势，包括强大的数据存储能力和并行处理能力、良好的可伸缩性和灵活性，使得基于云计算环境进行大规模图数据的高效处理，得到了数据库学术界和工业界的广泛关注，具有良好的应用前景。

利用云计算平台处理大规模图的迭代算法的相关研究工作是在近5到10年展开的，研究历程可以分为三个阶段：第一阶段，采用基于MapReduce模型[14]开发的Hadoop[11]作为

处理平台，并根据迭代处理的特点进行优化，如HaLoop[15]和Twister[16]等；第二阶段，开发基于BSP模型[17]的以图顶点为中心的专用大图处理平台，如Pregel[7], Hama[18], Giraph[8]等；第三阶段，以基于BSP模型的处理平台为基础，设计数据划分、消息通信的优化处理策略，进一步提高处理效率，如GPS[9]、PowerGraph[10]系统的主要贡献就在于此。

前两个阶段主要集中于迭代平台的设计与实现。由于MapReduce模型是一个数据流模型[7]，在迭代处理过程中需要引入较高的Warm-Up开销和数据传输开销，因此第一阶段的处理平台逐渐被第二阶段的平台所取代。BSP模型是一个状态模型[7]，避免了MapReduce模型的缺点，适合高频迭代的图处理算法。

目前的大图处理系统，在数据划分方面，主要采用第三方的图划分库，如GPS系统支持METIS库[19]，或者简单的随机Hash划分（RHP），如Pregel、Giraph、Hama等[7,8,20]。METIS算法库能够得到“高内聚、低耦合”的划分结果，减少通信开销，但是划分代价较高，而RHP方法虽然简单快捷，但是无法保证负载均衡且划分结果无局部性，导致迭代过程的通信开销较大。此外，对于大图处理，目前只有Hadoop、HaLoop和Hama[11,15,20]，支持磁盘操作。Hadoop与HaLoop受限于MapReduce模型的制约，效率较低[11,15]，而Hama只是简单的将消息推送磁盘[20]，没有设计相应的索引，磁盘操作开销极大。最后，关于消息优化方面，传统的消息合并（Combine）技术被Hadoop、Pregel等广泛采用[11,7]，能够有效处理高入度的大图，对于高出度的大图，主要有GPS的LALP技术[9]和PowerGraph的Vertex-Cut技术[10]。

1.3 本文主要贡献及组织结构

1.3.1 本文主要贡献

当前的大图迭代处理研究，主要处于第二个阶段，而第三阶段的工作，相对较少，即针对图的特点，进行优化处理。本文主要从分析大图增量迭代特点切入，从数据划分、磁盘索引和消息优化三个方面设计优化方案，具体贡献如下：

（1）详细分析了基于DFS技术和BFS技术爬取的原始输入图的局部性，采用连续划分策略，其数据划分效率优于随机Hash划分并且能够保留输入图的原始局部性，故迭代通信开销也低于随机Hash划分；

（2）设计了图顶点连续编码技术，可以将原始输入图顶点的字符串标识转换为数字连续编号，节省存储开销、网络通信开销，便于设计高效索引，提高处理效率；

（3）分析了图迭代过程的状态转换模型，提出了基于状态转换与Markov模型的动态可调Hash索引，以提高磁盘存取效率；

（4）分析了BSP同步模型和完全异步模型对于磁盘存取和消息规模的影响，提出了基于EBSP模型的Hybrid迭代机制，支持消息跨步剪枝（ASMP）与跨步合并（ASMC）

优化。

（5）基于连续划分策略，引入Vertex-Cut技术，提出了VCCP消息优化方案，可以通过较低的预处理开销，显著降低消息通信规模。

（6）设计实现了DiterGraph原型系统，支持大规模图的分布式磁盘迭代处理。

1.3.2 本文组织结构

本文一共分为七章，章节安排如下：

第1章为引言部分。首先介绍了云计算和大数据的应用背景，然后介绍了增量迭代的概念，在此基础上，提出了大规模图的增量迭代问题，分析了其特点与挑战并简单总结了国内外研究现状，最后提出了本文解决的问题并总结了本文的贡献点。

第2章是相关工作。首先简单介绍了大图迭代处理的计算框架，然后从数据划分、磁盘索引和消息优化三个方面，阐述了当前已有的相关工作。

第3章主要介绍本文的连续划分策略和图顶点连续编码技术。通过分析DFS生成图和BFS生成图的局部性，本文设计了连续划分策略，其划分代价低于目前普遍采用的随机Hash划分，但是能够保留图的原始局部性，所以通信规模小于随机Hash划分。但是连续划分策略要求图顶点必须为数字连续编号，否则通信过程无法定位消息，所以本文设计了图顶点连续编码方案。

第4章是磁盘存储格式设计和索引技术。首先介绍了大图增量迭代过程中的状态转换模型，然后提出了基于列存储模型的静态Hash索引，设计了图数据和迭代中的消息数据的存储格式，最后介绍了基于状态转换与Markov模型的动态可调hash索引，优化本地图数据的磁盘存取开销。

第5章为消息优化技术，包括两部分：第一部分为基于EBSP模型的Hybrid迭代机制，支持消息跨步剪枝和跨步合并策略；第二部分为基于连续编码的VCCP技术，能够以较低的划分代价，获得较好的消息优化效果。

第6章对DiterGraph原型系统进行了简单介绍，主要包括系统的功能模块介绍、部署及使用方法和可视化管理工具。

第7章是全文总结和对未来工作的展望。

第2章 大图迭代计算的相关工作

2.1 大图迭代计算的分布式框架

2.1.1 基于 MapReduce 模型的计算框架

Google 于 2004 年介绍了其分布式编程思想 MapReduce[14]。MapReduce 是一个数据流模型，用户需要自定义 Map 任务和 Reduce 任务的处理逻辑。

2008 年，作为 MapReduce 模型的开源实现，Hadoop 成为 Apache 的顶级项目[11]，吸引了 Yahoo、淘宝等大型网络公司的注意。Hadoop 以其良好的可扩展性和强大的容错能力，已经成为大数据处理的默认标准平台。

通过编写链式 MapReduce 作业，可以利用 Hadoop 的扩展性与容错能力，轻松实现大图的迭代处理。但是 Hadoop 是为处理“单趟”、“易并行”应用而开发的通用的数据流处理平台，面对具有强耦合性的高频迭代图处理算法，具有如下缺点：

（1）Warm-Up开销：Hadoop采用“申请式”任务调度机制，各计算节点申请需要处理的任务，开销较大，加上作业初始化、任务初始化等预处理工作，即使是一个空负载的Hadoop作业，也需要25-30秒的开销，因此，高频迭代的Hadoop作业将引入巨大的Warm-Up开销；

（2）静态数据操作开销：所谓静态数据，即迭代过程中状态保持不变的数据，比如PageRank算法中的图拓扑结构信息，在Hadoop中，这部分数据需要反复与HDFS交互并在Shuffle阶段进行网络传输和本地存取，引入极大的数据存取开销和网络通信代价；

（3）容错控制复杂：Hadoop强大的容错控制能力，仅限于一个Hadoop作业，而图迭代处理需要连续启动Hadoop作业，两个作业之间的容错控制，需要用户自己控制；

（4）数据的存储依赖于HDFS，难以设计高效的磁盘索引；

（5）Shuffle阶段的默认排序操作，增加了额外开销；

（6）编程接口差：Hadoop是通用计算平台，图迭代计算过程中的收敛性检测、图算法的表达，均需要转化为Map和Reduce函数，不易实现。

因此，直接使用基于 MapReduce 模型的 Hadoop 平台实现图的迭代处理，效率低下。作为改进，HaLoop 在继承 Hadoop 优势的同时，针对迭代处理进行了专门的优化[15]。HaLoop 将迭代过程的静态数据与动态数据分离，静态数据可以缓存至本地磁盘。缓存数据块的位置记录在主控节点，即 JobTracker。HaLoop 的 Loop-Aware 任务调度机制，尽量将新任务调度到缓存数据所在的计算节点，以避免静态数据在网络间的反复传递。考虑到随着迭代的收敛，需要处理的本地静态数据将逐渐减少，HaLoop 对本地磁盘驻留的

静态数据建立索引，以避免无用数据的存取开销。此外，HaLoop 提供了良好的迭代编程接口，如最大迭代步数的设置、迭代收敛的检测机制等。但是 HaLoop 仍然需要反复启动 Map 任务和 Reduce 任务，当缓存数据块所在的计算节点负载已满时，Loop-Aware 机制失效，必须另外指定节点运行该任务，此时仍存在静态数据的迁移开销，而其磁盘索引机制，依赖于 Shuffle 排序的结果且是静态机制，无法随着迭代状态的变化而动态调整。

Twister 是 Hadoop 的深度改进版本[16]，与 HaLoop 不同的是，Twister 假设数据全部驻留内存，任务进程也通过缓冲池保存，避免了数据序列化、反序列化开销和 Warm-Up 开销。但是 Twister 的底层存储系统并不完善，用户需要手动切分输入数据，加上数据全部驻留内存，因此实用性差。

2.1.2 基于 BSP 模型的计算框架

BSP 模型，即 Bulk Synchronous Parallel，“大块”同步模型，其概念由哈佛大学的 Valiant 和牛津大学的 Bill McColl 提出，是一种异步 MIMD-DM 模型，支持消息传递系统，块内异步并行，块间显式同步[17]。该模型采用主从式结构，基于一个 Master 进行全局协调调度，所有的 Worker 同步执行任务。BSP 模型通过全局同步，将两次迭代的计算处理以及消息分离。

Pregel 是 Google 提出的基于 BSP 模型的大规模图迭代处理系统[7]。Google 使用 Pregel 完成 20%的处理任务，而剩余的 80%的工作由 MapReduce 完成。Pregel 是一个以图顶点为中心的基于内存的系统，每次迭代称为一个“超级步”。本文后续章节中，将不区分“超级步”、“步”，均指一次迭代操作。一个图处理作业被 Pregel 分为多个任务并行迭代执行。各任务首先从数据源加载数据至本地内存，然后通过随机哈希分割完成图的划分操作。在之后的每个超级步中，每个任务调用各自维护的图顶点，执行用户自定义的 compute 函数，处理上一个超级步接收的消息，更新顶点的值并根据出度边发送新的消息。两个超级步之间，通过全局同步分离。Pregel 通过 checkpoint 机制进行容错控制，一旦某个超级步发生故障，各任务回滚至上一次的 checkpoint 处，重新开始迭代。

Apache 的 Hama 和 Giraph 是 Pregel 的开源克隆[8,20]。Hama 最新版本支持消息数据的直接磁盘存取，但是无索引机制的磁盘操作，开销巨大[20]。而 Giraph 是基于 Hadoop 框架开发的内存系统[8]，一个 Giraph 作业，即一个 only-map 的 Hadoop 作业。Giraph 通过 Map 任务中的内置循环来模拟 BSP 模型的迭代过程。与 Pregel 和 Hama 不同的是，Giraph 支持迭代过程中的数据以数据块为单位动态调整，以实现负载均衡。此外，Giraph 将每个作业的作业控制中心分布到不同的计算节点，避免单点瓶颈。但是 Giraph 以目的图顶点为单位组织发送消息，如果发送阈值设置过大，则各顶点的消息难以达到阈值，导致大量消息在本地计算结束后集中发送，网络阻塞；如果阈值过小，则会频繁启动发

送过程，导致建立通信连接的代价迅速上升。同时，在消息接收端，由于多个发送源可能同时发送消息给目的图顶点，因此，需要频繁的同步加锁操作，影响处理效率。

GPS[9]是斯坦福大学开发的大图处理系统，数据驻留内存。GPS 扩展了图处理的 API，提供 *Master.compute* 函数用于提供全局逻辑操作。对于数据的初始划分，GPS 分析了各种划分策略，如简单的 RHP 方法和复杂的 METIS 方法库[19]，默认采用简单随机 Hash 方法。在动态负载均衡方面，与 Giraph 不同，GPS 支持以顶点为单位的重划分策略。GPS 采用 LALP 方法对超大出度边切分，以减少消息发送规模。但是 LALP 方法假设出度边不参与计算，因此无法支持单源最短路径计算等应用。而且划分阈值是出度边的绝对长度，而不是分布在某个分区上的规模，因此无法给出合理的阈值以保证收益。

2.1.3 其它分布式计算框架

卡内基梅陇大学开发的 PowerGraph[10]系统设计了 GAS 编程模型，支持同步处理和异步处理机制。PowerGraph 采用 Vertex-Cut 方法降低高出度图对于处理性能的影响。Vertex-Cut 方法是在初始数据划分时完成，PowerGraph 提出了三种实现方案：基于随机划分的 Vertex-Cut（本文称为 VCRHP）方法，Oblivious 方法和 Coordinated 方法。三种方法均可以保证负载均衡，虽然预处理时间是递增的，但是划分的局部性也随之增强。划分的局部性越强，顶点的平均备份数目就越少，迭代过程中的网络通信规模就越少。

Trinity 是微软开发的图数据管理系统[21]，支持图数据库查询和复杂迭代处理。Spark 是加州大学伯克利分校开发的大数据处理系统[22]，提供 RDD 数据结构，拥有丰富的操作原语，支持流数据操作，可以模拟支持 MapReduce 作业和 Pregel 作业。Maiter[5]系统则基于累加迭代分析，支持异步迭代处理。上述系统中，除 Spark 外，均为内存处理系统，但是 Spark 的磁盘索引是简单的静态机制，与 HaLoop 类似，无法随着迭代状态的变化而动态优化。

2.2 大图的分布式划分

对于分布式图处理系统，必须有一个图数据加载器，将原始图数据从磁盘上加载到集群中。在加载过程中，需要将输入图 G 划分到 k 个任务上，以便于分布式处理。将图 G 划分为 k 个均衡的子图（顶点数目均衡或者边数目均衡），并且各个子图之间被切分的边的数目最小，即“高内聚、低耦合”，是一个 NP 难问题[23]。

2.2.1 随机 Hash 划分算法

RHP 算法，因其简单易行，目前被 Pregel、Giraph、GPS 和 Hama 等系统采用[7,8,9,20]。即各任务在数据加载时，根据图顶点标识符的值（比如 HashCode 值）与任务数目进行取模运算，决定其所在的任务。如果图顶点值的分布是完全随机的，则可以保证各任务

的图顶点数目的均衡性。但是，RHP 方法没有考虑局部性，划分完成后，各任务负责的子图之间，一般耦合性较强，会导致后续的迭代计算过程产生大量的消息通信。而划分本身，也需要对图数据全局交换洗牌，通信开销较大。此外，由于真实图的出度边存在幂律分布特点，所以出度边数目难以保证均衡，严重影响负载均衡。进一步的，如果图顶点标识符的值是字符串，则其 HashCode 值的分布无法保证随机性，那么各任务的图顶点的数目也难以达到均衡。

2.2.2 启发式划分算法

很多著名的启发式划分算法可用于图划分，如METIS算法库[19]、PMRSB[24]和Chaco[25]等，均具有较好的划分效果，能够显著降低子图之间的耦合性并保证均衡性。但是，这些算法的时间复杂度无法衡量且多是为科学计算而开发。Isabelle Stanton等人提出了10种启发式图划分算法并通过实验评估了所有算法的划分效果，但这些算法都是集中式的，面对大图，其划分效率较低[23]。

2.3 大图的磁盘存储与索引技术

由于图数据具有半结构化、属性无模式（Scheme-free）等特点，传统的组织存储模式，无法对其进行有效管理。而复杂图算法的高频迭代，更是对海量图数据的磁盘存储提出了挑战。目前广泛应用的关系数据库系统，模式固定，灵活性和可伸缩性较差，并不适合组织存储海量图数据。而大量关系表连接操作，也严重影响迭代计算的性能。新兴的图数据库系统，如Neo4J[26]等，多采用属性图模型组织图数据，由于避免了关系模型中的表连接操作而得到了性能的提升。但是现有的图数据库系统主要面向图数据的管理，如频繁的增删改查，对于支持复杂图应用的迭代计算过程，效率并不高。

在图迭代计算过程中，每个超级步需要反复对本地的图数据（图顶点和边）进行存取，以完成消息和本地数据的匹配、本地计算，计算结果也可能要求持久化。由于图算法迭代过程中，数据的访问具有频率高、局部性差等特点，如果数据驻留磁盘，会严重影响处理效率，所以Pregel、Giraph、GPS和PowerGraph等系统[7,9,10]，均假设数据驻留内存以回避该问题。

目前的大图处理系统，只有Hadoop、HaLoop和Hama支持磁盘存储[11,15,20]。Hadoop的存储系统包括HDFS和各任务的本地磁盘。使用Hadoop完成图的迭代处理，邻接表数据将按照传统关系数据库的行式存储模式存放于HDFS，对于本地磁盘，则是以key-value对为基本单元的临时文件。HaLoop将原始图数据和迭代过程中产生的动态数据（即消息）存放于HDFS，而静态数据则驻留本地磁盘并建立索引。HaLoop的索引机制依赖于静态数据和动态数据的Shuffle排序，建立代价较高，而且索引机制是静态的，无法随着图的

迭代收敛而动态优化，影响了索引效果。Hama目前仅支持消息接收端将溢出的消息数据简单推送磁盘存储，本地计算时再从磁盘读取处理。由于没有高效的数据管理方法和索引机制，一旦消息数据驻留磁盘，Hama的运行效率将迅速降低。

2.4 大图处理的消息优化方法

消息是分布式并行系统中重点研究的方面，是并行系统中的“阿喀琉斯之踵”，这也是Pregel等系统，目前只支持稀疏图的重要原因之一。消息的优化处理，主要是减少图迭代过程的消息规模，加快消息的传播。

2.4.1 同步迭代处理

虽然图的增量迭代处理本质上不需要全局同步，但是基于BSP模型的同步迭代处理机制可以显著约减消息规模。因为在一个超级步中，发给同一个目的图顶点的消息将被缓存，直到所有图顶点接收到所有消息，才启动本地计算处理。所以一个图顶点在一个超级步中仅被处理一次，按照出度边发送一次消息，这可以避免频繁更新导致的多次消息发送。对于增量迭代，接收的消息可以立即合并，以减少消息缓存的规模。目前，Pregel、Giraph和Hama等系统[7,8,20]，均采用BSP同步迭代方式。

2.4.2 异步迭代处理

异步处理方式即图顶点收到消息后，即可启动本地计算并发送新的消息，而不必等待接收到所有入度顶点的消息，也不需要等待其它顶点的消息接收操作。显然，异步迭代可以加快消息的传播，提高迭代收敛速度。Maiter 即采用完全异步的处理方式[5]。但是异步迭代导致图顶点被频繁更新，产生大量的消息。即使数据全部驻留内存，当图的入度较大时，异步迭代的计算效率要低于 BSP 同步迭代方式[6]。如果数据驻留磁盘，消息达到的无序性和频繁的本地计算，将导致巨大的磁盘存取开销。

2.4.3 基于 Combine 的消息优化方法

图的增量迭代算法中，消息具有可合并性，即发往同一个目的图顶点的消息可以合并，这可以显著降低网络通信规模和存储开销。Pregel、Giraph、GPS[7,8,9]等图处理系统以及 Hadoop[11]等通用云计算系统，都采用了该策略。Combine 操作对于高入度的图顶点，能够显著降低消息规模。进一步的，GPS 在系统测试时发现[9]，如果消息发送与本地计算异步进行，由于发送端的消息缓存有限，而消息的到达具有无序性，因而发送端的 Combine 操作对于消息的压缩比例有限，节省的网络通信开销将被 Combine 操作本身的遍历开销抵消。特别的，如果图较为稀疏（入度小），发送端的 Combine 操作会降低总体处理效率。因此，GPS 仅在接收端执行消息的合并处理。此外，如果完成本地计

算之后，集中发送消息，则发送端可以缓存所有消息，能够提高 Combine 的压缩比例，但却增大了发送端的存储开销，且集中发送时，会导致网络阻塞，严重影响效率。

2.4.4 基于切分的消息优化方法

真实世界的图一般具有幂律分布特点，某些图顶点的出度或者入度会远远超过其它顶点。高入度的情况可以通过 Combine 操作解决。对于高出度的顶点，GPS 和 Trinity 等[9,21]将其出度边切分，发送到边的目的顶点所在的分区，并在目的分区备份该顶点。本地计算时，仅需发送一条消息给目的分区的备份顶点，然后备份顶点在本地转发，可以减少消息的网络通信规模。

卡内基梅陇大学开发的 PowerGraph 系统[10]，使用 Vertex-Cut 技术，将所有顶点的出度边分布式存放，并将源顶点“切分”成多份（即备份），分别在不同的目的分区管理部分出度边。该过程是在数据划分时完成的。虽然 PowerGraph 的 Vertex-Cut 机制，能够有效降低迭代过程的消息通信规模，但是数据划分阶段的预处理开销较大。

2.5 本章小结

本章主要介绍了大图迭代计算的相关工作：总结了当前主要的大图迭代处理系统的特点并分析了其不足；介绍了大图划分的 RHP 方法和高级启发式划分算法；描述了 Hadoop、HaLoop 和 Hama 的大图磁盘存储格式以及索引机制；最后从迭代机制、入度边和出度边角度，总结了当前的消息优化处理方案。

第3章 分布式图划分与顶点连续编码技术

3.1 大图的分布式划分

3.1.1 大图的局部性分析

目前的大图分析处理，主要面向网页链接分析和社交网络分析。典型的应用，诸如 PageRank 计算，连通域计算，最短路径查询等。而原始输入图，一般是通过网页爬虫或者社交网站提供的 API 接口（如新浪微博数据访问接口）爬取得到。能否获取高质量的原始图数据，直接决定了后续分析处理结果的准确性和有效性。因此，在过去的十几年间，很多研究者致力于分析如何从真实世界中，抽取出高质量的图数据[27-31]。其中，深度优先遍历（DFS）和广度优先遍历（BFS）是两种经典的数据抽取方法。尤其对于 BFS 方案，Marc Najork 和 Janet L. Wiener 等人指出，该方案可以生成高质量的图数据[29]。此外，面对当前爆炸性增长的网络数据，并行爬取技术，也成为研究热点。而 BFS 算法，以其优异的并行性，获得了很多研究者的关注[31]。

大图的分布式处理，首先需要解决的问题就是将原始输入图切分为|*P*|个分区，以供|*P*|个任务并行计算处理。为便于分析图的切分问题，我们首先通过定义 3.1 介绍大图分布式处理的出度边局部性概念。

定义 3.1 出度边局部性（OELP）。假设原始输入图 G 以邻接表格式组织，将其切分为|*P*|个分区，也即|*P*|个子图 $G_1, G_2, ..., G_p$，每个子图的顶点集合边集记为 V_i 和 E_i，且有 $\bigcap_{i=1}^{P} V_i = V$，$\bigcap_{i=1}^{P} E_i = E$，$\forall i, j$，$1 \le i < j \le |P|$，$V_i \bigcap V_j = \varnothing$，$E_i \bigcap E_j = \varnothing$，则在并行处理过程中，$\forall e \in E_i$，其中 $e = (v_s, v_d)$，如果 $v_d \in V_i$，则称边 e 满足局部性（OELP）。

显然，满足 OELP 特点的边的数目越多，在迭代处理过程中，通信规模就会越低，处理效率将得到提高。而通过 DFS 技术或 BFS 技术获取的原始输入图，其部分出度边，满足 OELP 定义（见图 3.1 和图 3.2）。

在图 3.1 中，以顶点 A 作为出发点，进行深度优先遍历，得到的拓扑结构如左图所示，对应的邻接表如右图所示。然后将其邻接表顺序切分为 4 个分区（即 4 个子图）。如果输入图是一棵有向树（实线构成的出度边），则顶点 B、F、J 和 K 的分支，可以保留在同一个分区内，其所在分支内的所有出度边，均满足 OELP 特点。在这种情况下，通信代价，只有边(A, F)，(A, J)，(A, K)。然而，现实世界的顶点连接关系通常十分复杂，不满足有向树的定义（即存在图中虚线构成的边）。为描述真实世界有向图的 OELP 特性，我们引入两个定义：前驱边集（定义 3.2）和后继边集（定义 3.3）。

定义 3.2 前驱边集（PEC, predecessor edge collection）。图 G 以连接表存储，在 G

的抽取过程中，假设V_{done}为已经爬取的图顶点集合，对于新加入V_{done}的图顶点 v，其出度边为$E^{out}[v]$，则$\forall e \in E^{out}[v]$，有 $e=(v_s, v_d)$，如果$v_d \in V_{done}$，则 e 属于顶点 v 的前驱边集，即$e \in PEC[v]$。

定义 3.3 后继边集（SEC, successor edge collection）。显然，$SEC[v]=E^{out}[v]-PEC[v]$。

在图 3.1 中，顶点 E 的边(E, B)，顶点 G 的边 (G, B)和(G, D)，均为前驱边集，前驱边集的目的顶点所在的分区，具有随机性，如边(E, B)，满足 OELP 特点，而(G, B)和(G, D)则不满足 OELP 特点。若图 G 的平均出度为 m 且某出度边 e 所连接的子图的深度优先生成树的深度为 k，那么，该边所连接的子图的邻接表组织格式的最大存贮规模（图顶点与出度边的目的顶点）为$2\cdot(m^k-m)/(m-1)$，只要该值小于一个分区的最大存储规模，那么对于该子图内的顶点 v，$\forall e \in SEC[v]$，均满足 OELP 约束，我们称 DFS 生成图的这种局部性为**垂直局部性**。综上所述，在 DFS 生成图中，造成通信开销的出度边可以归纳为两类：

（1）前驱边集中的部分出度边，如(G, B)和(G, D)；

（2）后继边集中，用于连接不同分区内的子图的出度边，如(A, F)，(A, J)和(A, K)。

G 中其余的出度边，均满足 OELP 特点，不会产生通信开销。

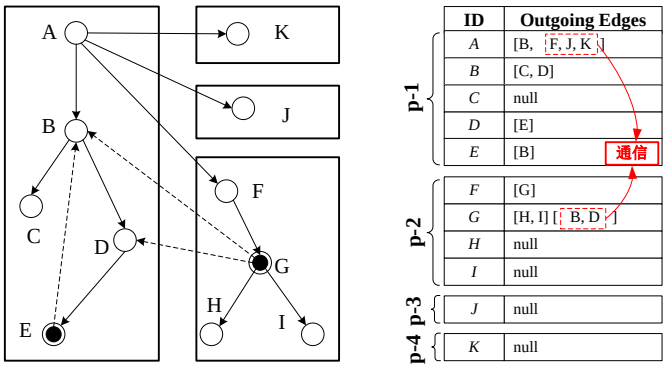


图 3.1 DFS 生成图的 OELP 特性分析

Fig. 3.1 Analysis of OELP for a DFS-generated graph

图 3.2 展示了 BFS 生成图的特点，与 DFS 生成图类似，边(C, D)和(H, D)均属于前驱边集，目的图顶点的分布具有随机性。与 DFS 生成图的垂直局部性不同，BFS 生成图具有**水平局部性**。也即，同一个顶点所有出度边，倾向于分配到同一个分区内，如图 3.2 中的边(A, B)，(A, C)，(A, D)和边(H, I)与(H, J)。对比后继边集中的跨区边，可以发现，对于 BFS 生成图，隶属于同一个源顶点的出度边，其分布具有密集性。例如，顶点 B 的出边(B, E)和(B, F)，其目的顶点均位于同一个分区，特殊情况下，会出现分割存储，如顶点 C 的出边(C, G)和(C, H)，其目的顶点分布在两个不同的分区内。这种密集性，是 DFS 生成图所不具备的，在 5.2 小节中，我们将利用这一特性，进行消息优化剪枝处理。

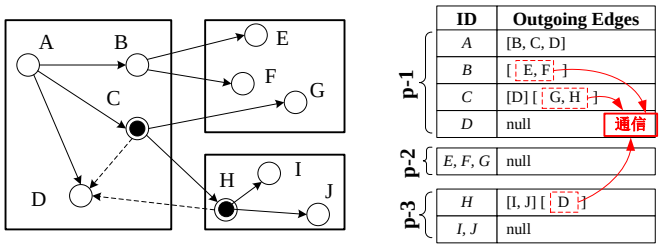


图 3.2 BFS 生成图的 OELP 特性分析

Fig. 3.2 Analysis of OELP for a BFS-generated graph

3.1.2 连续划分方法

当前主流的分布式图处理系统，如 Pregel，Hama，Giraph 和 GPS 等，均采用 RHP 方法切分图数据的邻接表。RHP 算法，具有三个缺点：（1）无法保证均衡性；（2）无法保留图本身的局部性；（3）划分过程中，需要大量通信开销。在具体分析上述三个缺点之前，我们首先定义计算负载（定义 3.4）概念。

定义 3.4 计算负载（CompLoad）。图的分布式迭代计算，主要包括消息接收、本地计算和消息发送，则对于分区 P_i，计算负载 $CompLoad(P_i)$ 的公式见 3.1：

$$(|V_i|+|E_i^{out}|)\cdot Cost_{lc}+|E_i^{in}|\cdot Cost_{in}+|E_i^{out}|\cdot Cost_{out}+|E_i^{out}-E_i^{OELP}|\cdot Cost_{net} \quad (3.1)$$

其中，V_i 为分区 P_i 的顶点集合，E_i^{out} 为出度边集合，E_i^{in} 为入度边集合，E_i^{OELP} 为满足 OELP 特性的出度边集合，$Cost_{lc}$ 为本地计算开销，$Cost_{in}$ 为消息接收处理开销，$Cost_{out}$ 为发送端消息的组织管理开销，$Cost_{net}$ 为消息发送过程的网络通信开销。若 $|E_i^{out}|\approx|E_i^{in}|$，则公式 3.1 可简化为公式 3.2：

$$CompLoad(P_i)\approx|V_i|\cdot Cost_{lc}+|E_i^{out}|\cdot(Cost_{lc}+Cost_{in}+Cost_{out})+|E_i^{out}-E_i^{OELP}|\cdot Cost_{net} \quad (3.2)$$

对于 RHP 算法，如果图顶点 ID 的 HashCode 值的分布是随机的，则可以保证各个分区的 $|V_i|$ 的均衡性，否则就会存在哈希偏斜。进一步的，即使能够保证图顶点数目的均衡性，由于真实世界的图，其出度边一般存在 Power-Law 偏斜规律，故难以保证出度边数目的均衡性，导致计算负载的不均衡。此外，在 3.1.1 小节已经分析，原始图数据具有一定的局部性，而 RHP 方法，仅根据图顶点 ID 的 HashCode 值进行数据分割，完全破坏了原有的局部性，这会导致后续迭代过程中，分区之间的消息通信量增大。最后，RHP 过程需要根据取模函数的结果，决定该条记录所属的分区并发送至目的分区，相当于对原始数据进行了重新分配，这种全局洗牌会引入大量的消息通信开销。

鉴于 RHP 方法存在以上三个缺点，本文设计了连续划分方法（CP 方法）。假设原始图数据存放于分布式文件系统 HDFS 上，CP 方法根据 HDFS 提供的 API，将输入数据切分为|P|个字节数相同的逻辑 split 分片，每个 split 分片记录了该分片对应的原始数据的起始偏移量、总字节长度、数据存放的物理节点等信息。然后任务调度模块按照数

据本地化和负载均衡（各个工作节点执行的任务数目均衡）的原则，为每一个任务指定一个 split 分片并将该任务分配到一个工作节点上运行。各任务启动之后，首先根据各自的 split 信息，加载部分原始图数据至本地，同时收集该部分数据的统计信息。对于第 i 号任务，其统计信息 I_i 为：$I_i =< ID[task],|V_i|,|E_i| >$，其中 ID[task]代表任务编号，而$|V_i|$和$|E_i|$表示图顶点的数目和出度边的数目。数据加载完毕后，各任务将自己的本地数据统计信息 I_i 发送给作业控制中心，JobInProgress 汇总统计信息并以此为基础，将原始输入图切分为$|P|$个分区，第 i 个分区的数据，也即第 i 号任务上所加载的数据，所以每个分区的元数据信息，与各任务汇报的统计信息 I_i 一致。

3.1.3 连续划分方法分析

相比于 RHP 方法，CP 方法具有三个优势：（1）保证分区之间计算负载的均衡性；（2）保留图的原始局部性，（3）数据划分过程无网络通信开销。

CP 方法直接保留本地加载的图数据。由于本地加载的数据规模由 split 分片决定，而各个任务的 split 分片的字节规模是均衡的，因此各分区加载的图顶点数目和出度边数目的总和是相对均衡的。

RHP 和 CP，均属于 Edge-Cut 方法[10]，会导致部分出度边被跨区切分，引入消息通信开销。对于 RHP 方法，Joseph E. Gonzalez 等人[10]给出了被跨区切分的边的数学期望值为$(1-1/|P|)\cdot|E|$，其中$|E|$为总出度边数目。由于 CP 方法可以保留 DFS 生成图的**垂直局部性**和 BFS 生成图的**水平局部性**，因此切分的出度边数学期望值可由公式 3.3 得到：

$$(1-\frac{1}{|P|})\cdot(|PEC|+\alpha\cdot|SEC|)=(1-\frac{1}{|P|})\cdot\beta\cdot|E| \tag{3.3}$$

其中$0\le\beta\le1$，因此 CP 方法的通信代价低于 RHP 方法。β 的具体取值与图的拓扑结构特征相关，在实验环节，我们将具体分析 β 参数的取值。

CP 方法使各任务能够保留各自加载的图数据，不需要 RHP 方法的全局洗牌过程，因此划分过程无通信开销。而 RHP 方法，其划分过程的总通信规模可由公式 3.4 估算：

$$Comm(RHP)=(1-\frac{1}{|P|})(|V|+|E|) \tag{3.4}$$

其中，$|V|$和$|E|$代表输入图的总顶点规模和总出度边规模，$|P|$是分区数目。

3.2 图顶点连续编码技术

真实世界的原始输入图，其顶点 ID 通常具有一定的物理意义，比如网页拓扑图，顶点通过网页的 URL 唯一标识，而对于路网数据，顶点的标识符则是地标名称等。在 3.1 小节的 CP 划分策略中，为实现后续迭代计算的消息通信寻址，要求图顶点 ID 必须

为数字连续编号。这就需要提供图顶点连续编码功能，将原始输入图中顶点 ID 的字符串，转换为数字连续编号。除此之外，对比于字符串格式表示的图顶点 ID，采用数字连续编号技术还具有如下优势：

（1）节省存储空间，降低数据序列化和反序列化操作开销。

（2）降低消息通信开销。消息的一般表示格式为二元组 M=<DstID, MsgValue>，即目的顶点 ID 和消息值。如果 DstID 为数字，则消息通信的字节规模会降低，改善网络通信性能及因通信引入的消息序列化和反序列化操作开销。

（3）提高匹配查找效率。在本地计算过程中，需要通过图顶点 ID 完成消息数据和目的图顶点之间匹配操作，采用数字连续编号后，其匹配效率将远高于字符串匹配操作。

（4）便于内存使用的评估与分配。因字符串的长度不一致，难以准确估算图数据和消息数据的内存使用比例，影响内存分配的准确性。根据统计信息计算 ID 的平均长度，进行分配，难以消除内存溢出的问题。而数字连续编号，因其格式统一，便于估算内存，可提高内存使用率，避免内存溢出现象。

（5）连续编号的图数据，以邻接表格式存储，便于建立各种高效的索引。

假设原始图按照邻接表格式，采用行式存储，存放于磁盘，欲将图中顶点的原始标识符统一替换为连续编号的数字标识符，其处理流程主要包括三个阶段：

（1）顶点统计与编号分配：各任务并行加载原始图数据，统计各自负责的图顶点规模，为每个任务分配一个编号范围；

（2）顶点编码：各任务根据分配的编号范围，在本地完成图顶点字符串的数字连续编码，构造原始顶点标示符与数字编号之间的映射关系表 *MT*；

（3）编号替换：各任务查找 *MT* 表，将出度边中的图顶点字符串替换为对应的数字连续编号。

在给定的计算集群上，完成大规模图的分布式连续编码工作，其挑战主要集中在如何维护 *MT* 表和高效的完成出度边的编号替换工作。

对于大规模图，*MT* 表的规模是及其可观的。以网页 URL 为例，实际应用中，如 Google，通常检索 400 多亿网页，而互联网中已经存在的 URL 总数，超过万亿级别。如此大规模图的 *MT* 表，需要 TB 规模的内存才可以支撑，在有限的集群资源环境下，*MT* 表不可能全部驻留内存。*MT* 表需要各个任务进行分布式的磁盘存储维护。此外，出度边中，目的图顶点的分布具有随机性，因此，在编号替换阶段，会构成对 *MT* 表的随机查询请求，导致频繁的磁盘 I/O 操作。

为高效的完成图顶点数据的连续编码，本文设计了两种实现方案。第一种方案利用 Hadoop 完成该功能，简单易行，但是处理效率低。另一种方案，基于分布式 Hash 表技术（DHT），考虑数据本地化和负载均衡等原则，通过设计高效的磁盘存储格式，有效

提高了编码效率。

3.2.1 基于 Hadoop 的连续编码方法

Hadoop 是一个通用的基于磁盘的分布式处理框架，为完成顶点连续编码，共需要三个 Hadoop 作业，分三个阶段完成如下工作：图顶点编号范围分配与连续编码，替换出度边中的目的顶点，还原出度边。如图 3.3 所示，第一个作业将启动相同数目的 map 任务和 reduce 任务，Map 阶段，读取原始图数据信息，统计各个 map 任务负责处理的图顶点的规模并创建编号分配索引，通过设置 Partitioner，可以保证 map 任务的输出数据仅被传送到任务号相同的 reduce 任务中，例如，map_0 任务的输出，仅由 reduce_0 处理，“消除了”shuffle 过程中的数据交互操作。Reduce 阶段，各任务读取索引时，按照顺序递增求和的方式计算其图顶点的起始编号，完成本任务图顶点的顺序编号。例如，reduce_n 将读取索引中的 map_0 至 map_(n-1)，共计（n-1）项的顶点数目，累加计算，得到本任务顶点起始编号。*MT* 表由各个 reduce 任务输出至 HDFS 上存储。

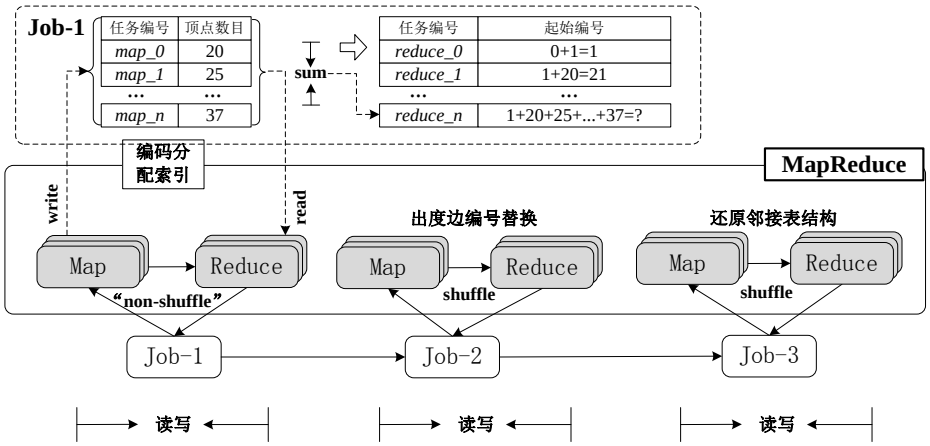


图 3.3 基于 Hadoop 的连续编码

Fig. 3.3 The Hadoop-based consecutive encoding method

第二个作业的 Map 阶段，读取 *MT* 表和出度边数据，完成切分操作。在 Reduce 阶段，利用局部的 *MT* 表，完成出度边的目的顶点编号替换，其输出结果为完成替换的出度边。第三个作业的 Map 阶段，读取出度边表，在 Reduce 阶段，转化为原始的邻接表，最后输出完成连续编号操作的图数据。

图 3.4 和图 3.5 以网页 URL 的连续编码为例，分别介绍了出度边的编号替换和邻接表的还原组织过程。如图 3.4 所示，在第二阶段的 Hadoop 作业中，首先在 map 任务中顺序读取数据并进行顺序处理。对于任意一条编号完毕的图数据，例如（1，URL1，URL3 URL5），Map 过程将其拆封为两种输出类型。一种是输出的 Key 为 URL1（即当前图顶点的原始顶点值），Value 为（1,1），即类型为 1、URL1 的编号为 1 的值对。另一种是输出的 Key 分别为当前图顶点的出度边的目的顶点值，例如 URL3、URL5 等，Value 为

（2,1），即类型为 2、它的入度顶点编号（是 URL1，编号为 1）。在 Shuffle 阶段，MapReduce 将相同 Key 的所有键值对映射到一个 reduce 任务中，在映射后的 Key 的值列表中，必然存在 2 种类型的 Value，一种是类型为 1 的 Value，仅一个值，一种是类型为 2 的 Value，有多个值。在 Reduce 阶段，系统将一个 Key 下的 Value 列表中，类型为 2 的值抽取出来作为输出的 Key，输出的 Value 均为类型为 1 的值。比如将 Key 为 URL1 的值列表中，（2,3）和（2,4）中的 3、4 抽取出来作为 Key，将（1,1）中的 1 抽取出来作为 Value，最终的输出结果为（3,1）、（4,1）等。

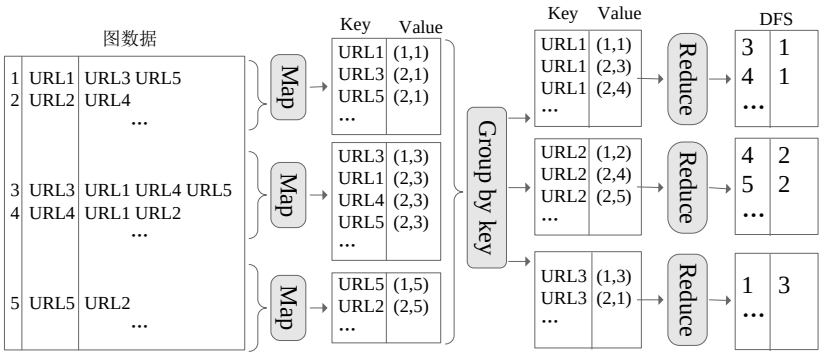


图 3.4 出度边目的顶点的编号替换过程

Fig. 3.4 Illustration of encoding destination vertex ids of outgoing edges

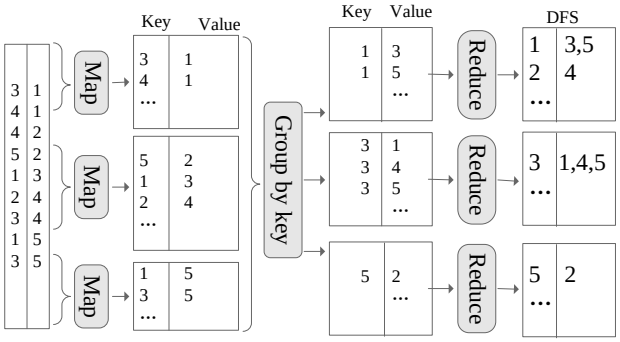


图 3.5 邻接表的还原过程

Fig. 3.5 Illustration of recoverying the adjacency list

第二阶段的输出结果，将作为第三阶段的输入。如图 3.5 所示，map 任务并行读取完成编号替换的出度边并将输入的键值对直接输出。在 Shuffle 阶段，MapReduce 将相同 Key 的所有键值对映射到同一个 reduce 任务中，即把源顶点相同的各出度边映射到一个 Value 列表中。在 Reduce 阶段，系统将一个 Key 下的 Value 列表中的值进行合并，输出结果的 Key 不变，输出结果的 Value 被赋予合并后的值，即出度边列表。

3.2.2 基于 DHT 的 Hybrid-*MT* 连续编码技术

基于 Hadoop 的连续编码策略，简单易行，但效率较低（详见 3.2.3 小节分析），而且必须作为预处理操作，提前完成，与 CP 划分策略难以兼容。因此，本文提出了基于

DHT 的 Hybrid-*MT* 连续编码策略，解决上述问题。

图 3.6 展示了连续编码的执行流程。当一个图处理作业的若干任务完成 CP 划分之后，各任务将向 JobInProgress 汇报各自的图顶点数目，构造类似图 3.3 中所示的编码分配索引表，然后该索引表将发送至所有任务，各任务依据索引表完成本地图顶点的连续编码，构造出 *MT* 表。为提供高扩展性，支持大规模图的编号替换操作，*MT* 表可以被递归 Hash 分割，最终保证每个哈希桶内的 partial-*MT* 能够一次性加载到一个任务的内存中。在递归分割 *MT* 表的过程中，JobInProgress 将建立一个递归分割树，也即分割索引，记录切分过程。当分割索引的所有叶子节点对应的哈希桶中的 partial-*MT* 的规模，均小于单个任务的内存时，递归分割过程结束。分割索引将会被所有任务复制到本地，并以此为基础，对本地的出度边数据，进行 OE-MERGE 操作，即将需要替换的目的图顶点按照分割索引合并到一个哈希桶内，以便于集中替换，减少磁盘 I/O 开销。同时，完成 OE-MERGE 后，可以统计出分割索引中所有叶子节点的替换请求负载，JobInProgress 以此为依据，综合考虑数据本地化和负载均衡等指标，将每个叶子节点分配给唯一的任务。分布在各个任务上的叶子节点的编号映射数据，按照任务映射关系完成迁移整理工作之后，即为替换索引。最后，作业控制中心统一调度，各个任务批量异步加载部分替换索引至内存，直到完成所有出度边的目的顶点的编号替换工作。

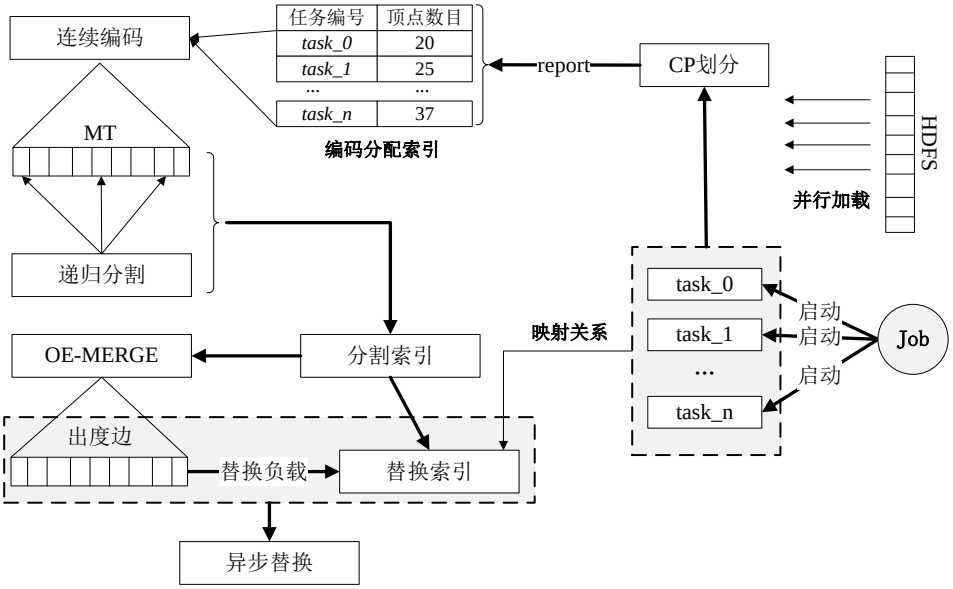


图 3.6 基于 DHT 的连续编码

Fig. 3.6 The DHT-based consecutive encoding method

各任务根据编码分配索引完成本地连续编码之后，形成图顶点原始标识符与编码之间的映射表 *MT*，*MT* 的每条记录的格式为<StrID, CodeID>。之后，应该通过查找 *MT* 表，将出度边中所有目的图顶点标识符替换为对应的 CodeID。鉴于 *MT* 表是由各个任务分布式维护，故该问题是一个分布式环境下的索引及寻址过程。由于出度边的规模十分庞大，通常达到几十亿甚至上百亿的级别，如何根据每一个出边顶点的原始标示符而快速

找到其全局编号所在的位置并完成替换，是一个挑战。

为提高替换查找时的定位效率，本文利用哈希索引技术，将 *MT* 表按照 StrID 进行哈希分割，索引 *MT* 表的所有记录。假设选取的哈希函数为 $h_i = hash(StrID_i)$，$1 \le i \le N$，对任意 $i \in [1, N]$，有 $h_i \in [h_{\min}, h_{\max}]$，可以得到 N_h 个 Hash 桶。对任意一个任务 *n* 来说，它处理的 partial-*MT* 中的记录可能被散列到任意的 Hash 桶 *p* 中。反之，每个任务中，*p* 桶中的记录都只是全局 Hash 桶 *p* 所包含的所有记录中的一部分。这种 Hash 桶的管理方式，称之为**垂直管理模式**。在出度边的替换过程中，垂直管理模式，可以保证本地出度边能够借助 Hash 索引而快速完成出度边顶点的编号替换。然而对于本地匹配失败的出度边顶点，则需要向所有任务广播查询请求。各任务接到请求后，执行本地查询，如果查询成功，则返回查询结果，否则丢弃该请求。垂直管理模式，对本地替换的局部性要求很高，否则，会造成大规模的通信开销和无效的本地查询处理，严重影响效率。

为消除垂直管理模式对局部性的严重依赖，本文基于分布式哈希表（DHT）技术，提出了**混合管理模式**（Hybrid-*MT*）。Hybrid-*MT* 方法将各个任务分别维护的哈希桶内的部分数据，进行归并集中，某一个桶内的数据，仅由一个任务维护，这可以避免远程替换时的消息泛洪。同时，各任务的本地数据仍然保留，以便高效完成本地替换过程。也即，在 Hybrid-*MT* 管理模式中，出度边顶点首先在任务内部完成本地替换，过滤后的出度边顶点，根据 Hybrid-*MT* 路由向特定的任务发送查询请求。

对于 Hash 索引桶，理想情况下，每个 Hash 桶的 hash 值长度为 $L_h = {(h_{\max} - h_{\min})}/{N_h}$，第 *p* 个 Hash 桶范围是 $h_p \in [(p-1)L_h, pL_h]$，其中 $1 \le p \le N_h$。但是，Hash 处理通常存在严重的数据偏斜，会导致各个桶的 partial-*MT* 的规模相差很大，一方面，加大了查询请求处理过程中发生负载偏斜的可能性，另一方面，由于 Hybrid-*MT* 方法会将某个桶的数据集中管理，可能导致某些桶由于规模过大，超出单个任务的可用内存容量，在这种情况下，出度边顶点的随机替换请求，会导致严重的磁盘 I/O 开销。因此，在执行 Hybrid-*MT* 管理之前，需要提供 Hash 桶的递归切分功能，确保每个桶内的数据规模，不超过单个任务的内存容量，避免额外的磁盘 I/O 开销，控制负载均衡的偏斜。

图 3.7 展示了递归分割的过程以及用于记录递归分割过程的分割索引树。递归分割的过程，由作业控制中心统一调度，但分割过程，由各任务并行完成，因为此时的 MT 表，是由各任务分布式维护的。图 3.8 中举例说明了对于 2 号 hash 桶的递归分割过程。假设单个任务的可用内存容量为 150MB，1 号桶分割之后，其子桶包括 2-5 号，其中 3-5 号桶内的数据规模均小于 150MB 的切分阈值，故直接放入 Finish-Queue 队列，而 2 号桶的数据规模达到 300MB，需要放入 Wait-Queue 队列，等待被重新切分。当作业控制中心调度 2 号桶时，各任务得到通知，开始并行切分本地 2 号桶内的数据。task_1 将其

正常切分为 3 个子桶并统计各子桶的数据规模。task_2 和 task_3 分别说明了两种特殊情况。task_2 中，该任务并没有 2 号桶内的数据，因此，虽然 task_2 接到了递归切分命令，但是并无实际的切分操作。task_3 虽然存在 2 号桶的数据，但是切分后的子桶中，7 号桶为空。各任务切分完成后，分别向作业控制中心汇报切分结果。而 task_2 和 task_3 的汇报内容，需要“补齐”缺失的子桶，对于缺失桶，其规模容量设置为 0，表征该子桶在本任务中，无数据。作业控制中心汇总了累加各任务的汇报结果，得到子桶（6,7,8）的全局容量，并根据汇报数目判定是否切分完毕。也即，每个子桶，都应该接到 3 个汇报结果（总共有 3 个任务）。这也是 task_2 和 task_3 需要“补齐”缺失桶的原因。当所有子桶均已得到 3 次汇报后，2 号桶的递归切分结束，从 Wait-Queue 队列中移除。作业控制中心对子桶进行判定，6 号桶的数据规模达到 200MB，超出阈值，放入 Wait-Queue 队列，等待递归切分，而 7 号和 8 号桶，则直接放入 Finish-Queue。当 Wait-Queue 队列为空时，即结束递归切分过程，此时，分割索引的所有叶子节点（也即 Finish-Queue），其对应的 Hash 桶的数据规模均小于切分阈值。

MT 表完成递归切分后，如果直接按照 Hybrid-*MT* 方法完成叶子节点到任务之间的分配，然后执行出度边的替换操作，存在两个缺点：（1）分配叶子节点时，只能根据叶子节点的数据规模，估算其查询处理负载，而真实图中一般存在 Power-Law 偏斜，以有向图为例，也即入度偏斜，故各叶子节点的查询负载并不与其数据规模成正比，因此，分配时，可能导致各任务的处理负载不均衡；（2）出度边的分布具有随机性，而叶子节点无法同时驻留内存，如果直接遍历出度边发送替换请求，会导致大量的随机替换请求，造成目的图顶点所在的叶子节点频繁的进行内外存交换，开销巨大。

因此，在执行 Hybrid-*MT* 方法之前，需要对出度边数据，按照递归分割索引进行整理。假设原始图数据的邻接表格式为<StrID, $DstStrID_1$, $DstStrID_2$, …>，则图顶点连续编号之后，变为<CodeID, StrID, $DstStrID_1$, $DstStrID_2$, …>。出度边的整理操作，是以源顶点为单位完成，对于某个源顶点 StrID，遍历其所有的出度边，按照分割索引进行切分，隶属于同一个叶子节点（b-*i*）的边，作为一个 EdgeBlock，格式为<CodeID, $DstStrID_1$, $DstStrID_2$, …>，其中，CodeID 标注该 EdgeBlock 中的边所属的源顶点，以便替换完成后，进行邻接表的还原。出度边整理结束后，各任务分别汇报各哈希桶的查询负载（即分配到各桶的出度边的总数目）并由作业控制中心汇总，完成各叶子节点查询负载的精确统计（见图 3.8 中的 load）。由于在 Hybrid-*MT* 方法中，各任务所包含的 partial-*MT* 的内容，仍然保留，因此，在出度边的整理完成后，首先在本地完成本地替换。即，如果出度边目的图顶点的编号可以在本地查找到，则直接替换，否则，分配到对应的哈希桶中，等待全局替换。最终汇报的 load，应该去除本地替换命中的出度边数目。

如图 3.8 所示的网页 URL 数据，其中，urla 的出度边<urla, urlb>, <urla, c>，urlb 的

出度边<urlb, urle>，和 urlc 的出度边<urlc, urlf>，按照分割索引，均属于第 1 个哈希桶，因此，$EdgeBlock_1$=<1, urlb, urlc>, $EdgeBlock_2$=<2, urle>和 $EdgeBlock_3$=<3, urlf>均放入 1 号桶存储管理。

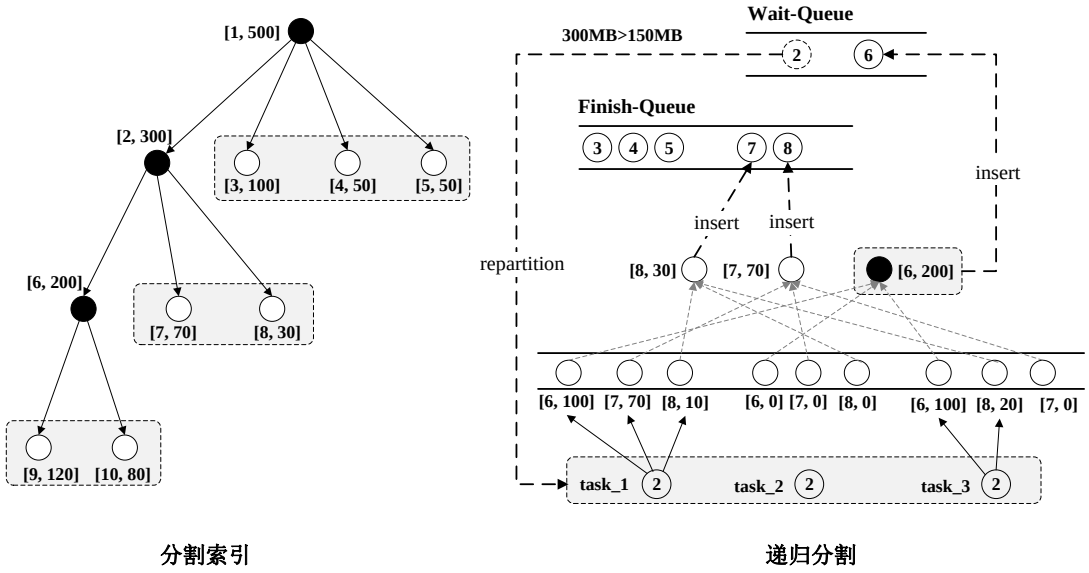


图 3.7 分割索引与递归分割

Fig. 3.7 Partition Index and the process of recursively partitioning

此时，第 i 个叶子节点的元数据信息如下：L_i=<id, size, load, taskId, list<e>>，e=(taskId, taskSize, taskLoad)，其中，id 为该叶子节点的哈希桶编号，全局唯一；size，为该叶子节点对应的哈希桶内的总数据规模；taskId，为最终组织维护该叶子节点的所有数据的任务编号，此时为空值；第四项，则用于记录该桶内数据及负载的分布，即编号为 taskId 的任务，包含的该桶内部分数据的规模为 taskSize，同时，对该桶的查询负载为 taskLoad，所谓的查询负载，即需要请求该哈希桶提供对应的顶点编号的出度边的数目。taskSize 的分布，是在递归分割过程中记录得到。

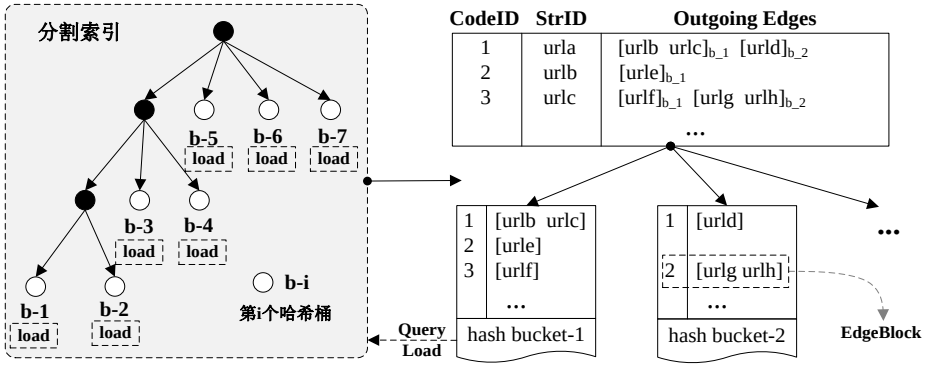


图 3.8 分割出度边

Fig. 3.8 Illustration of partition outgoing edges

出度边整理完成后，作业控制中心开始将分割索引中的叶子节点分配给具体的任务并执行编号替换工作，即执行 Hybrid-*MT* 方法，建立替换索引。假设|*P*|个任务处理能力

相同，而 *L* 个叶子节点的处理开销，使用 L_i.*load* 表示，各叶子节点的处理，相互独立且具有原子性（处理时不可中断，不可拆分成子问题），则该问题是一个多机调度问题，即将 *L* 个独立的工作分配给|*P*|个任务完成，使总处理时间最短。多机调度是一个 NP 难问题，因此，一般使用贪心算法求出近似最优解[32]。Hybrid-*MT* 方法分配叶子节点时，在贪心算法的基础上，还需要考虑如下原则：

（1）数据本地化：每个叶子节点对应的哈希桶内的数据，一般都分散存贮在|*P*|个任务中，如果将第 *i* 个桶的所有数据集中由第 *i* 个任务处理，就必须将分散在其余|*P*|-1 个桶内的数据远程传送至第 *i* 号任务，造成网络通信开销。数据本地化原则，即将第 *i* 个桶分配到 L_i.*list* 中，taskSize 值最大的任务，以最小化网络传输的数据规模。

（2）查询本地化：选择 L_i.*list* 列表中，taskLoad 值最大的任务作为维护该叶子节点数据的实体，则该任务的出度边对于该哈希桶的查询，变为本地查询，可以减少 taskLoad 规模的网络通信量。

算法 3.1（见表 3.1）展示了将 N_{leaf} 个叶子节点组合成 N_t 个集合的贪心分配过程。首先将叶子节点队列按照每个叶子节点的负载，也即 load 值，从大到小排序（第 1 行）。然后，使用数组 *loadSum* 记录每个集合内的叶子节点的总负载（第 2-4 行）。之后，遍历叶子节点队列，对每一个叶子节点，将其分配到当前总负载最轻的集合中，并更新该集合内的总负载（第 5-10 行）。算法 3.1 通过贪心选择，可以得出一个近似最优的组合分配方案，降低各叶子节点完成编号替换过程的总处理时间。完成贪心组合之后，需要进一步将 N_t 个叶子节点的组合，分配到 N_t 个任务。分配过程中，需要考虑数据本地化和查询本地化的原则，减少网络通信开销。

表 3.1 叶子节点的贪心组合优化

Table 3.1 Greedily combinatorial optimization of *Leaf Nodes*

算法 3.1 *greedyLeaf(LeafNodeQueue, N_t)*

输入：叶子节点队列 *LeafNodeQueue*，任务数目 N_t

输出：叶子节点组合结果集合 *C*

```
1.  sort LeafNodeQueue by leaf_i.load;
2.  loadSum = creatArray(N_t);
3.  for i = 0 to N_t - 1
4.      loadSum[i] = 0;
5.  for i = 0 to LeafNodeQueue.size() - 1
6.      leaf = LeafNodeQueue.removeFirst();
7.      min = getMinIndex(loadSum);
8.      loadSum[min] += leaf.load;
9.      put leaf into C_min; // C_min: 第 min 个叶子节点的组合
10.     update(C, C_min); // 更新 C 中的第 min 个元素的值
11. end
```

分配流程见算法 3.2（见表 3.2）所示。由于图顶点数据和出度边数据的分布没有必然的关联，因此数据本地化和查询本地化有可能产生矛盾的结果，即各自选择的最优任

务不同。考虑到图顶点数据和出度边的查询替换，均执行一次，因此，使用两者的加和作为综合衡量指标。具体的，对于某个叶子节点的 list 列表，假设为该叶子节点选择的任务的编号为 id，每条查询的字节流量为 byte，则其图顶点数据迁移和查询替换导致的网络通信开销可以利用公式 3.5 估算：

$$f(list,id)=\sum_{\forall e\in list\wedge e.taskId\neq id}(e.taskSize+byte\times e.taskLoad) \tag{3.5}$$

以 $f(list,id)$ 作为综合评价指标，算法 3.2 展示了 Hybrid-*MT* 方法的优化分配策略。

表 3.2 数据本地化与查询本地化优化算法

Table 3.2 The algorithm of optimizing data-locality and query-locality

算法 3.2 *localityOptimal(C,N_{leaf}, N_t)*

输入：叶子节点组合结果集合 *C*，叶子节点数目 N_{leaf}，任务数目 N_t

输出：叶子节点组合的任务分配结果 *R*

```
1.  M = new Matrix(N_leaf, N_t); // 创建一个拥有 N_leaf 行、N_t 列的矩阵
2.  rowIndex = 0;
3.  foreach C_i ∈ C
4.    foreach leaf ∈ C_i
5.      foreach taskId ∈ taskIDs  // taskIDs: leaf 的 list 中所有任务编号的集合
6.        M[rowIndex][taskId] = f(leaf.list, taskId);
7.      end;
8.      rowIndex++;
9.    end;
10.   [taskId, b] = findMinCost(M, C_i); // b 为代价变量
11.   V_i = (C_i, taskId, b);
12. end;
13. while V ≠ ∅
14.   foreach taskId ∈ V.getTaskIDs()
15.     V_i = findMinCost(taskId, V);
16.     put <V_i.C_i, taskId> into R;
17.     V = V / V_i;
18.     erasure(M, taskId);
19.   end;
20.   foreach V_i ∈ V
21.     [taskId, b] = findMinCost(M, V_i.C_i);
22.     reset(V_i, taskId, b);
23.   end;
24. end;
```

在代价分析矩阵 *M* 中，M_{ij} 代表第 *i* 个叶子节点分配到第 *j* 个任务时的网络通信开销（第 3-12 行）。在计算矩阵 *M* 时，以叶子节点组合 C_i 为单位，通过调用 *findMinCost(M, C_i)*函数，为 C_i 内的叶子节点，选择一个综合代价最小的任务并记录在向量 V_i 中（第 10-11 行）。之后，对向量集合 *V* 中的元素进行循环处理。每次循环过程中，找出当前 *V* 中所有任务编号的集合，也即，这些任务，均为当前可能的最优解集合。对其中每个任务，调用 *findMinCost(taskId, V)*函数，从若干叶子节点的组合中，选择代价最小的组合 C_i，将 C_i 分配给该任务处理（第 15-16 行）。之后，将该向量 V_i 从集合 *V* 中删除，并将矩阵

M 中，第 taskId 列标记为擦除（即不可用）（第 17-18 行）。随后，更新集合 *V* 中的剩余向量，为尚未分配的叶子节点组合选择次优的任务并更新向量集合 *V*（第 20-23 行）。该过程直至向量集合 *V* 为空为止。*R* 中记录了每个叶子节点组合和其对应的任务的分配关系，作业控制中心据此调度 *MT* 表的分配，完成替换索引的建立过程。

在之后的编号替换过程中，作业控制中心统一调度每个任务加载一批叶子节点至内存，然后，对应哈希桶中的出度边数据开始发送消息，完成替换查询。整个替换过程，所有叶子节点仅加载一次。完成编号替换后的出度边数据，按照其源顶点编号进行整理，还原为邻接表格式。

3.2.3 顶点编号替换的代价分析

利用 Hadoop 完成图顶点的连续编码，需要启动三个作业，完成三个阶段的工作（详见 3.2.1）。整个过程需要对整体数据完成四次磁盘交互工作：三次作业的输入与输出，编码完成后，图数据导入分布式图处理系统的磁盘交互。此外，需要在 Map 阶段和 Reduce 阶段，完成三次 Shuffle 操作。而基于 DHT 的 Hybrid-*MT* 方法，也需要对整体数据完成四次磁盘交互操作，分别为：数据加载阶段，出度边整理（包括本地编号替换），编号替换，还原邻接表格式。对于 Hybrid-*MT* 方法而言，额外的操作为递归切割。因此，如果递归切割的开销，小于 Hadoop 的 Shuffle 开销，就可以证明 Hybrid-*MT* 方法的优势。

假设 *D* 为数据的初始输入规模，*T* 为任务数目，R_v 为图顶点数据在 *D* 中所占的比例，$\overline{v_{byte}}$ 为平均每个图顶点所占的字节数目（原始图顶点 ID 为字符串，长度并不统一）。

在 Hybrid-*MT* 方法中，由于哈希偏斜，需要对图顶点数据及其编号值，也即 *MT* 表，进行递归分割。哈希偏斜的分布规律，可以使用类 Zipf 分布来模拟[33]。则将原始数据 *V* 切分为 *n* 个哈希桶后，第 *i* 个哈希桶内的顶点数目由公式 3.6 得到[33]：

$$H_i^v = \frac{|V|}{i^z \cdot \sum_{j=1}^{n} \frac{1}{j^z}} = \frac{|V|}{i^z \cdot (\ln n + \gamma + \varepsilon_n)} \tag{3.6}$$

其中，γ 为调和级数（Harmonic series）的欧拉-马歇罗尼常数，而 $\varepsilon_n \approx 1/2n$。因此，对于分割索引中的每个父节点，其被分割之后，子节点中，前 *k* 个需要继续被分割的哈希桶内的图顶点总数的比例计算方式见公式 3.7：

$$R_{rep} = (\frac{1}{j^1} + \frac{1}{j^2} + ... + \frac{1}{j^k}) \frac{1}{\sum_{j=1}^{n} j^{-z}} = \frac{\sum_{j=1}^{k} j^{-z}}{\sum_{j=1}^{n} j^{-z}} = \frac{\ln k + \gamma + \varepsilon_k}{\ln n + \gamma + \varepsilon_n} \geq \frac{\ln k + \gamma + \varepsilon_n}{\ln n + \gamma + \varepsilon_n} \tag{3.7}$$

当 $z = 1$ 时，即为 Zipf 分布。为计算简单，后续计算均取 $z = 1$，则令 β 为递归切分数据占父节点数据的比例，则 β 值可由公式 3.8 计算：

$$\beta=\frac{\ln k+\gamma+\varepsilon_n}{\ln n+\gamma+\varepsilon_n}=\frac{\ln n+\gamma+\varepsilon_n+\ln k-\ln n}{\ln n+\gamma+\varepsilon_n}=1+\frac{\ln k-\ln n}{\ln n+\gamma+\varepsilon_n} \tag{3.8}$$

当 n 增大时，k 减少，β 也在减少，故 β 为单调递减函数。设分割索引的递归分割深度为 d，可知，d 由偏斜最严重的哈希桶的数据规模（即第一个哈希桶）决定，假设每次递归分割时，均将父节点分割为 n 个子节点，则递归分割结束条件由公式 3.9 给出：

$$\frac{1}{(\ln n+\gamma+\varepsilon_n)^d}\le\frac{1}{\Delta} \tag{3.9}$$

其中，Δ 为 *MT* 表的数据规模与单个任务的可用内存 M 的比值，即公式 3.10 所示：

$$\Delta=\frac{DR_v}{M}(1+\frac{byte(c)}{byte(v)}) \tag{3.10}$$

由公式 3.9 可计算出递归深度为：

$$d\ge\frac{\ln\Delta}{\ln(\ln n+\gamma+\varepsilon_n)}=\left\lceil\frac{\ln\Delta}{\ln(\ln n+\gamma+\varepsilon_n)}\right\rceil \tag{3.11}$$

由公式 3.8 和公式 3.11 可得，递归分割过程中，数据的磁盘交互规模的最大值为：

$$2(\beta+\beta^2+...+\beta^d)\cdot(M\cdot\Delta) \tag{3.12}$$

这里的最大值，是假设每次递归切分时，需要切分的数据规模占父节点数据规模的比例均为 β。由于实际切分时，父节点的数据规模单调递减，而切分的子桶数目均等于 n，故实际的比例值，肯定小于 β。进一步的，由于 $0<\beta<1$ 且 d 为正整数，故 $0<1-\beta^d<1$，所以由公式 3.12 可以得到：

$$2\frac{\beta(1-\beta^d)}{1-\beta}\cdot(M\cdot\Delta)<2\frac{\beta}{1-\beta}\cdot(M\cdot\Delta) \tag{3.13}$$

Boduo Li 和 Edward Mazur 等人分析了 Hadoop 作业在 Map 和 Reduce 之间的 Shuffle 阶段，数据的磁盘交互规模[34]。其中，Map 任务输出端的磁盘交互规模为：

$$2T\cdot\lambda_F(\frac{DK_m}{TM},M) \tag{3.14}$$

欲保证递归分割的开销低于 Map 任务输出端的磁盘开销，由公式 3.13 和公式 3.14 可得，需要满足如下不等式：

$$2\frac{\beta}{1-\beta}(M\cdot\Delta)\le 2T\cdot\lambda_F(\frac{DK_m}{TM},M) \tag{3.15}$$

由公式 3.8 知，β 是关于 n 的单调递减函数。因此，只要每个父节点递归切分的哈希桶的数目 n 满足公式 3.15 的要求，则可以保证递归切分的磁盘开销小于 Map 任务输出端的磁盘开销。因此，Hybrid-*MT* 方法的磁盘存取开销小于 Hadoop 的磁盘开销。而数据本地化和查询本地化减少了网络通信开销，贪心调度叶子节点的分配组合，也提高了 Hybrid-*MT* 方法的效率。

3.3 实验结果与分析

3.3.1 实验设置

实验集群由 33 台计算机组成，1 台主控节点，32 台计算节点，每台计算机的配置为：2 Intel Core i3-2100 CPUs，8GB 内存和 500GB 硬盘，磁盘转速为 7200RMP。所有计算机安装 Linux RedHat 6.0 操作系统并由 1 台 Gbps 带宽的交换机互联。实验过程中，为避免同一台计算机上不同任务之间的内存与磁盘争用影响，每个计算节点仅运行一个任务，每个任务的 JVM 配置 2GB 内存。

实验采用经典 PageRank 算法作为示例程序，迭代 10 步。PageRank 算法的实验数据集如表 3.3 所示，包括：（1）美国专利引用情况网络图 Patent[35]；（2）LiveJournal 社交网络图 Live-J[35]；（3）Wikipedia 的网页链接图 Wiki-PP[36]；（4）Twitter 社交网站的拓扑图 Twitter[37]。原始图数据以邻接表格式存放于 HDFS，顶点的初始 PagRank 值为 10，出度边无权重。

表 3.3 实验数据集说明

Table 3.3 Characteristics of data sets

数据集	顶点	出度边	平均出度	文件大小
Patent	3,774,768	16,518,948	4.38	244MB
S-LJ	4,847,571	68,993,773	14.23	576MB
Wiki-PP	5,716,808	130,160,393	22.77	1.05GB
Twitter	41,700,000	1,470,000,000	35.25	12.6GB

3.3.2 图划分性能评估

实验从数据划分时间、负载均衡、通信规模和整体运行效率四个方面综合评估了 RHP 和 CP 方法的性能。

数据划分时间。数据划分时间包括数据加载和重划分两个过程。实验采用 10 个计算节点，启动 10 个任务测试。从图 3.9 可知，CP 方法的数据划分时间小于 RHP 方法。因为 RHP 需要对输入的图数据进行全局洗牌，网络通信开销大，而 CP 方法仅在调整负载均衡时引入少量通信。在 Patent、S-LJ 和 Wiki-PP 三个数据集上的测试结果表明，RHP 方法的运行时间分别是 CP 方法的 1.07 倍、1.85 倍和 2.12 倍。即图的规模越大，CP 方法的优势越明显。

负载均衡。本实验采用 Twitter 数据集进行负载均衡实验。Twitter 图数据集的出度边分布具有明显的幂律偏斜分布特点[38]。实验采用 32 个计算节点，启动 32 个任务测试。从图 3.10 可知，由于输入图已经采用数字连续编码，所以 RHP 方法可以保证各个分区的图顶点数目的严格均衡性，而 CP 方法是按照各分区负责处理的数据的字节规模划分，

因此无法保证图顶点数目的均衡。图 3.11 说明了各个分区的出度边数目的分布。由于出度边直接决定消息的生成与发送操作，因此出度边的偏斜将导致后续迭代计算过程中消息处理的负载不均衡。虽然 RHP 方法保证了图顶点数目的均衡，但是由于出度边分布的偏斜，导致各个分区中最高出度边数与最低出度边数的绝对差值高达 1272 万，而标准差为 301 万。对于 CP 方法，除最后一个分区因字节数目较少而出度边较少外，其它分区的出度边总数的分布大致均衡。如果除去最后一个分区，则最高分区与最低分区的边数绝对差为 568 万，标准差为 151 万。而图顶点数目与出度边总数之和，则决定了一个分区的总体计算负载。如图 3.12 所示，对于 RHP 方法，由于其图顶点数目严格均衡，故最高分区与最低分区绝对差值仍为 1272 万，标准差为 301 万；而对于 CP 方法，从图 3.10 和图 3.11 对比可知，图顶点数目和出度边数目呈互补趋势，即出度边多得分区，其图顶点数目较少，反之亦然，我们仍去掉最后一个分区，则最高分区与最低分区的绝对差值降为 31 万，标准差降为 55 万。

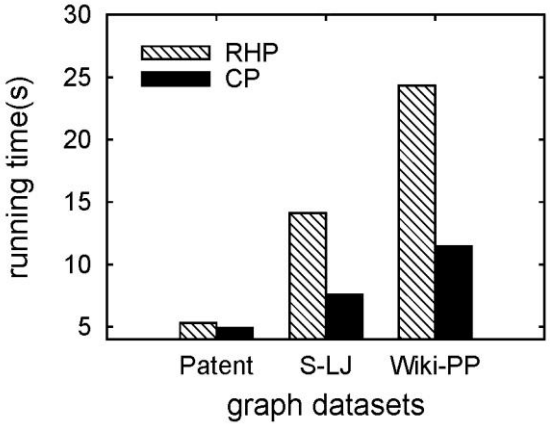


图 3.9 数据划分时间

Fig. 3.9 The running time of data partitioning

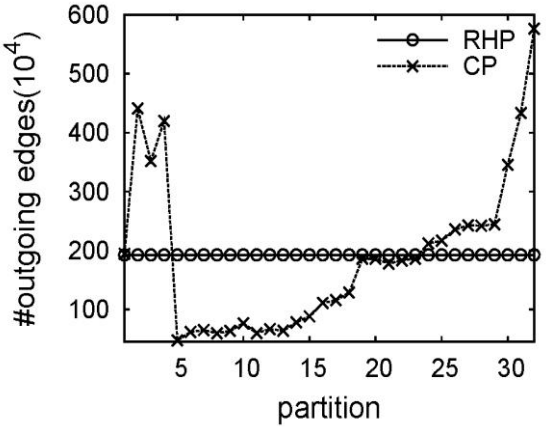


图 3.10 图顶点分布

Fig. 3.10 The distribution of vertices

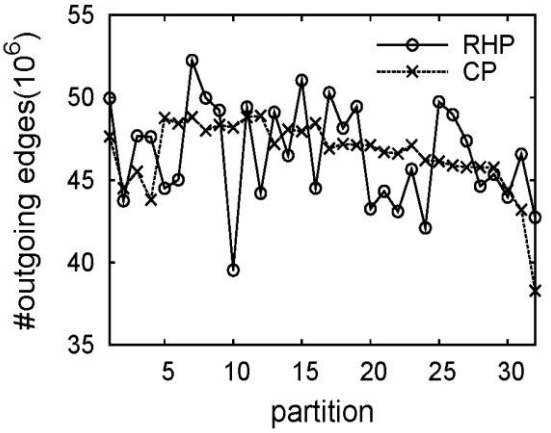


图 3.11 出度边分布

Fig. 3.11 The distribution of outgoing edges

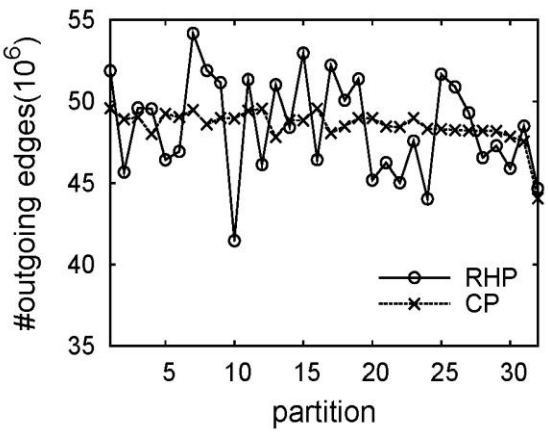


图 3.12 图顶点与出度边的综合分布

Fig. 3.12 The distribution of vertices and outgoing edges

局部性分析。根据 3.1.3 小节的分析，CP 划分中被跨区切分的边的数学期望为 RHP 方法的 β 倍，其中 $0 \le \beta \le 1$。而跨区切分的边即网络消息，因此网络通信规模（消息数

目）可以反映β的取值。由于经典 PageRank 算法中每个超级步的消息规模是相同的，因此图 3.13 对比了一个超级步中，RHP 方法和 CP 方法的网络通信规模。由于 CP 方法能够保留图的原始局部性，因此 CP 方法相比于 RHP 方法，可以减少网络通信量。对于数据集 Patent、S-LJ 和 Wiki-PP，β值分别为 93%、75%和 80%。β值基本与图的平均出度成正比，但是受图的具体拓扑影响（如 S-LJ 和 Wiki-PP，虽然前者平均出度低，但β值却略低于 Wiki-PP）。

总体运行时间。总体运行时间即作业从提交开始至迭代结束，包括：数据划分、迭代计算和保存结果三个阶段。如图 3.14 所示，RHP 方法的总体运行时间约为 CP 方法的 6 至 8 倍。这是因为 CP 方法的数据划分性能、负载均衡性能和局部性等指标均优于均优于 RHP 方法。

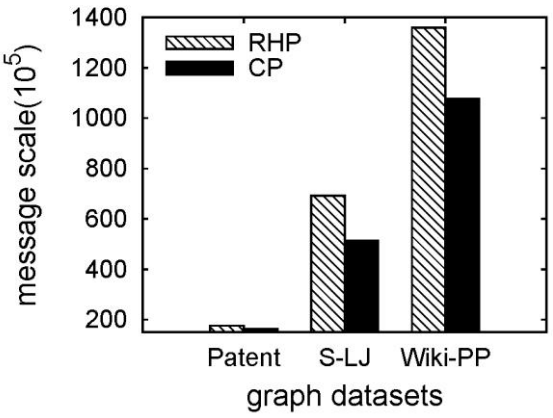


图 3.13 局部性分析

Fig. 3.13 Analysis of locality

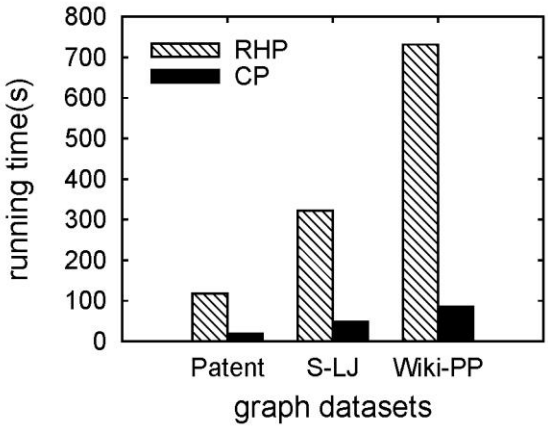


图 3.14 总体运行时间

Fig. 3.14 The overall running time

3.4 本章小结

本章主要介绍了原始图数据的局部性、图的划分方法和图顶点的连续编码技术。DFS 生成图的垂直局部性和 BFS 生成图的水平局部性，均有助于降低迭代处理时的通信开销。然而，图的 RHP 划分方法，虽然简单易行，却无法保留原始图的局部性。此外，RHP 方法容易造成计算负载的偏斜，导致水桶效应明显，且在划分过程中，需要全局 shuffle 图数据，引入巨大的网络通信开销。因此，本文提出了 CP 划分方法，解决 RHP 方不足，其在划分开销、局部性保留和负载均衡方面，性能优异。但是，CP 方法必须保证图数据的顶点是数字连续编号，否则无法完成迭代过程中的消息路由寻址操作。为解决原始输入图的顶点 ID 为字符串的情况，本章提出了基于 Hadoop 的连续编码方案和基于 DHT 的 Hybrid-*MT* 连续编码方法，对图数据进行连续编号。

第4章 基于状态转换与 Markov 模型的磁盘索引技术

4.1 大图的增量迭代特点分析

Yanfeng Zhang 等人给出了累加迭代的形式化定义和收敛性证明[5]。在累加迭代的基础上，本文针对图计算处理，给出图迭代处理的增量迭代计算模型：

$$\begin{cases} v^k = v^{k-1} \oplus \Delta v^k \\ \Delta v^{k+1} = \sum\limits_{v_i \in E_k^{in}[v]} \oplus(\Delta v_i^k) \end{cases}$$

其中，v^k 表示顶点 v 在第 k 步的值，由其在第 k 步的值与第 k 步的增量和 Δv^k 进行 $\oplus$ 操作得到。$\oplus$ 是用户自定义逻辑运算，与累加迭代相同，即必须满足交换律、结合律，且对数值 0 存在恒等性。Δv^k 是第 *k-1* 步顶点 v 的入度顶点发送的消息值的增量和。也即，顶点 v 在第 *k+1* 步的增量，是由其入度顶点在第 k 步的增量值累加得到。

4.1.1 经典算法分析

大量的图迭代算法可以使用增量迭代计算模型表达，比如单源最短路径计算，PageRank 计算，连通域计算等。

对于单源最短路径计算，$\oplus$ 操作为“取小值”操作。在第 k 步迭代过程中，图顶点 v 对其在第 *k-1* 的值 v_{k-1} 和第 k 步收到的消息的“累加和” Δv^k 进行“取最小值”运算，如果 $v^{k-1} > \Delta v^k$，则 $v^k = \Delta v^k$ 并按照出度边，松弛所有出度边顶点（也即发送新的消息，Δv_j^k，$\forall \Delta v_j \in E^{out}[v]$），否则，$v_k$ 的值不变，也不会松弛出度边顶点。显然，如果 $\Delta v^k = 0$，顶点 v 不必启动本地计算流程。

经典的 PageRank 算法[4]，其图顶点 v 的 PageRank 值的计算公式为：$v^{k+1} = \sum\limits_{v_i \in E_k^{in}[v]} \oplus vote[v_i^k]$，$vote[v_i^k]$ 是顶点 v_i 在第 k 步，对顶点 v 的投票值，一般为 $vote[v_i^k] = \dfrac{PR(v_i^k)}{|E^{out}[v_v]|}$，即其 PageRank 值按照出度边均分。其计算过程中，顶点本身的 PageRank 值并没有参与 $\oplus$ 运算。因此不符合增量迭代计算模型。而改进后的算法[39]，每个图顶点均维护自己的 PageRank 值，其“投票值”为 $vote[v_i^k] = \dfrac{PR(v_i^k) - PR(v_i^{k-1})}{|E^{out}[v_i]|}$。则 $v^k = v^{k-1} \oplus \Delta v^k$，$\Delta v^k$ 即为各入度顶点“投票值”的累积和。这里的 $\oplus$ 即为加法操作。

如果顶点的 PageRank 值，没有发生变化，即 $vote[v_i^k]=0$，则不必发送消息。反之，如果 $\Delta v^k=0$，则该顶点也不必启动计算流程。

连通域算法的简单实现[6]，即为每个图顶点分配一个唯一数字编号（3.2 小节的顶点连续编码技术，可以保证输入图的顶点是数字连续编号），每步迭代，各顶点均根据出度边向邻居发送自己的顶点编号，而每个顶点的值，即为收到的编号中，值最小的编号。迭代结束后，值相同的顶点，属于同一个连通域。

4.1.2 增量迭代特征

图中的增量迭代算法，其消息值，即为各顶点按照出度边发送的“增量值” Δv_i^k。本节将分析增量迭代算法，在消息的关联完整性、合并性和激活顶点规模三个方面的特征。

定义 4.1 关联完整性。$\forall v_i \in V$，V 为图顶点集合，$E^{in}[v_i]$ 为 v_i 的入度边图顶点集合，在第 k 步，顶点 v_i 可以对其收到的消息进行计算处理，当且仅当 v_i 收到了 $E^{in}[v_i]$ 中所有入度边顶点发送的消息。针对同一个顶点的入度消息之间的关联性，称为消息的关联完整性。

显然，增量迭代算法，对于消息没有定义 4.1 中的约束。因此，可以设计多种迭代优化机制，比如 Yanfeng Zhang 等人提出了异步累加迭代模型[5]。而本文则在第 5 章提出了 EBSP 模型，其主旨思路为“图顶点同步更新，消息跨步处理”。详细内容，将在第 5 章介绍。

由于增量迭代的⊕操作，满足结合律，其消息值具有合并性。如果所有数据全部驻留内存，则对于每个图顶点，仅需要保存两个状态值：v^{k-1} 和 Δv^k，消息即来即合并，不必维护庞大的入度边消息，能够节省存储开销[5]。如果数据驻留磁盘，消息的合并性也有利于设计高效的磁盘管理算法，并且便于精确估计图顶点的消息内存占用。

从 4.1.1 小节介绍的三个列子中可以发现，在增量迭代过程中，处于激活状态的图顶点（即参与计算，也即收到消息的图顶点）的规模，在不断变化，我们称这种现象为图的状态转换。在 4.1.3 小节，将详细分析不用应用和不同数据集的状态转换特点，并据此提出了增量迭代的状态转换模型。如果图数据（图顶点数据和出度边数据）驻留磁盘，则状态转换模型可以用于减少无效数据的扫描开销，这是本章动态 Hash 索引策略的理论基础。

4.1.3 增量迭代的状态转换模型

对于不用的图应用算法，以单源最短路径和 PageRank 计算为例，其状态转换模型，可以分为两类。前者，其迭代的状态转换过程为：扩张态，稳态，收缩态；后者，状态

转换过程为前者的子集，为：稳态和收缩态。

对于单源最短路径计算，从源顶点开始，按照出度边逐步松弛，在迭代的初始阶段，大部分图顶点尚未被松弛。因此，一个图顶点，一旦收到消息，一般会更新自身的最小路径值并继续松弛出度边顶点。由于每个图顶点的出度边规模，通常大于 1，因而，该顶点的松弛，会导致更大规模的消息被生成并传播出去，进一步的，在下一个超级步中，会有更多得图顶点被激活。因此，如图 4.1(a)所示，在迭代的初始阶段，消息规模和处于激活状态的图顶点规模，将逐步增大，即为扩张态。当大部分图顶点处于激活状态时，消息规模将达到峰值。此时，各顶点不断交换最小路径值，因此，消息规模和激活状态的图顶点规模将保持在一个较高的水平，即图 4.1(a)中的稳态。当大部分图顶点已经找到最短路径值后，其收到的消息将被全部剪枝（因此收到的消息值，即可能的最短路径值，均比该顶点的值大）。因此，消息规模将逐渐减少，相应的，消息规模也会递减。此时，图的迭代，进入图 4.1(a)中所示的收缩态。

对于经典 PageRank 计算，由于不是增量迭代计算，因此，每次迭代，所有的图顶点均处于激活状态，均按照出度边发送消息，因此，如图 4.1(b)所示，其消息规模和激活状态的图顶点规模，一直保持不变，处于稳态。而对于增量 PageRank 计算，在迭代开始时，所有图顶点处于激活状态，发送消息。各顶点不断交换 PageRank 投票值，处于稳态。随后，图顶点的 PageRank 值，将逐渐稳定，不再发送消息，而消息规模的递减，会导致激活的图顶点规模逐渐减少，即进入收缩态，直至收敛，迭代终止，如图 4.1(b)所示。连通域计算的过程，与增量 PageRank 算法类似，仅有稳态和收缩态，这里不再赘述。

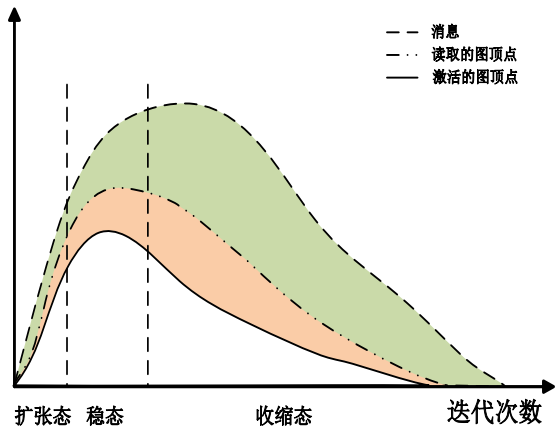


图 4.1(a) 单源最短路径计算

Fig. 4.1(a) The single source shortest path computation

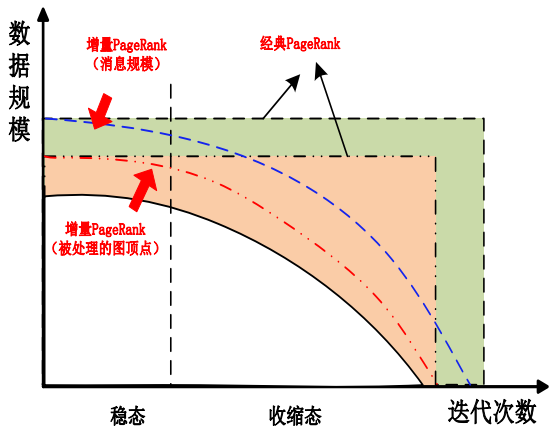


图 4.1(b) PageRank

Fig. 4.1(b) PageRank

由于单源最短路径包含全部三个状态，因此，本章后续内容，将以单源最短路径为例进行分析。图迭代处理过程中的三种状态各自具有不同的特点，可以采用不同的优化措施提高迭代处理效率。如图 4.1(a)所示，当图数据驻留磁盘时，由于数据必须按块读

取，“读取的图顶点”和“激活的图顶点”两条曲线之间的区域，表示该部分图数据被无效加载，增大了额外的磁盘存取开销，而“消息”和“激活的图顶点”两条曲线之间的区域，则表示消息的网络传输规模远大于实际接收消息的图顶点规模，而增量迭代中的消息又具有合并性，因此，消息的优化处理将是另外一个提高处理效率的方向。

在扩张态，消息规模和激活状态的图顶点规模均处于较低的水平，但是呈现上升趋势。在该阶段的优化措施，从磁盘存取效率方面，主要考虑如何有效组织图数据，避免对暂时尚不会处理的图数据的反复加载。在稳态，消息规模和激活的图顶点规模均处于较高的水平，基本上属于满负荷运行。在该状态的优化措施，应着重于有效管理大量消息和图数据的磁盘存取操作。收缩态的优化处理和扩张态相似，集中于提高磁盘存取效率。其中，关于扩张态和收缩态的磁盘优化策略，即本章重点介绍的动态哈希索引，而关于稳态阶段的消息优化，将在第 5 章介绍。

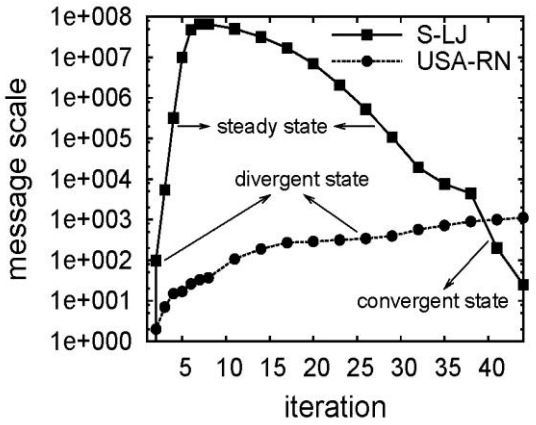


图 4.2(a) 每次迭代的消息发送规模

Fig. 4.2(a) The scale of sending messages per iteration

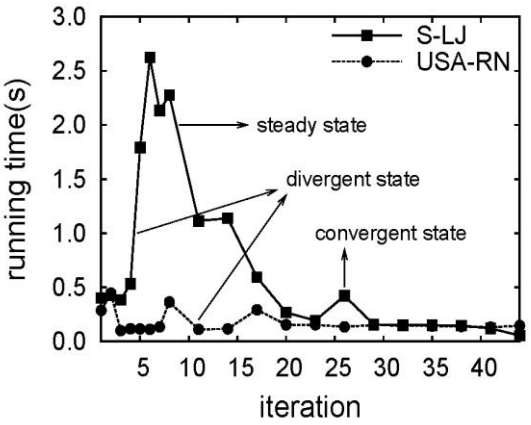


图 4.2(b) 每次迭代的运行时间

Fig. 4.2(b) The running time per iteration

虽然图的状态转换模型过程十分简单，但是不同状态之间，并没有明显的界限。这是因为不同图的拓扑结构，具有不同的特点。我们使用两个具有典型代表的真实图数据集测试了单源最短路径算法的状态转换过程。S-LJ 数据集是一个社交网络图，顶点之间联系紧密，而 USA-RN 则是美国路网图，顶点之间的联系相对稀疏。如图 4.2(a)与(b)所示，从消息规模和各步迭代处理时间分析，S-LJ 具有明显的“扩张态—>稳态—>收缩态”特征，而 USA-RN，其处于扩张态的时间远大于 S-LJ。在 4.2.2 小节，我们提出一种自底向上的方法来动态判定图的迭代状态。

4.2 大图的磁盘存储管理技术

图数据的迭代处理，具有数据访问频率高、局部性差等特点。因此，如果图数据和消息数据驻留磁盘，将会造成巨大的磁盘存取开销。本小节介绍基于列存储模型的静态哈希索引技术和基于 Markov 模型的动态哈希索引技术。

4.2.1 基于列存储模型的静态 Hash 索引策略

对于单源最短路径计算，假设$\delta[u]$为顶点 u 的最短路径值，$E^{out}[u]$为 u 的出度边列表，顶点 u 收到的所有消息 $msg[u]$使用 $lmsg[u]$表示。在迭代计算过程中，顶点 u 收到的消息，最终将被合并为一条消息，即 $msg[u]$，然后本地读取 u 的信息，即$\{u, \delta[u], E^{out}[u]\&w\}$，启动本地计算，如果$\delta[u] > msg[u]$，则更新$\delta[u] = msg[u]$并按照$E^{out}[u]\&w$松弛出度边顶点。

在 4.1.2 小节中提到，增量迭代的消息具有合并性，理论上，如果内存足够，所有图顶点都可以在内存中维护其$\delta[u]$和一条 $msg[u]$，则不存在 $lmsg[u]$。然而，如果上述条件无法满足，则只能有部分数据驻留内存，内存缓冲区溢出时，需将数据推送至磁盘上的临时文件存储。由于消息到达的无序性和较差的局部性，发往顶点 u 的消息，在不同的临时文件中，可能存在多个 $msg[u]$（但是，在一个临时文件中，最多只有一个$msg[u]$）。也即，如果数据驻留磁盘，$lmsg[u]$是存在的。

在整个迭代计算过程中，根据数据的访问特点，顶点 u 的所有相关数据$\{u, \delta[u], E^{out}[u]\&w, lmsg[u]\}$可以被分为三类：

（1）图顶点数据，$\{u, \delta[u]\}$：每次迭代过程中，如果顶点 u 的 $lmsg[u]$不为空，则 u 将被调用，执行本地计算，并根据用户自定义逻辑更新$\delta[u]$。

（2）出度边数据，$\{E^{out}[u]\&w\}$：如果图顶点数据被调用并执行本地计算，则其出度边数据也需要被提取，但是，出度边数据是只读的，不需要回写到磁盘。对于 PageRank 和连通域计算等，出度边数据均是只读的。

（3）消息数据，$\{lmsg[u]\}$：消息数据需要不断的按照目的图顶点进行合并。

消息数据，图顶点数据和出度边数据（后两者统称为图数据），按照列存储模式管理，在磁盘上分别存放，以减少无用数据的存取。在列存储模式下，消息数据的合并管理，与本地图数据无关，而本地计算时，由于图顶点数据和出度边数据的分离，在更新图顶点数据时，可以避免回写出度边数据。

列存储模型，可以有效避免无效数据的存取，提高效率。但是，图顶点的计算过程，本质上要求提取整行数据，则不同列之间的元组重组，会导致连接操作，影响效率。由于图数据驻留本地磁盘，因此，在数据加载阶段，可以将图顶点数据和出度边数据放在两列的同一行，如图 4.3 所示，可以实现图顶点数据与出度边数据之间的顺序定位，减少重组开销。但是，由于消息数据到达的无序性和较差的局部性，消息数据与图顶点数据之间，需要进行连接操作。采用图顶点连续编码策略后，本地图数据是按照图顶点编号顺序存放。一种可能的优化方案，是将无序到达的消息数据，按照目的图顶点编号进行排序，提高连接操作的局部性。但是，消息是动态生成和消耗的，即每次迭代，均需

要对接收到的消息进行基于磁盘的排序操作，代价高昂。Boduo Li[34]等人提出使用哈希方式，代替 MapReduce 在 Shuffle 阶段的排序操作，可以显著提高效率。因此，我们设计了一种静态哈希索引方式，提高消息数据与图顶点数据之间连接操作的局部性。

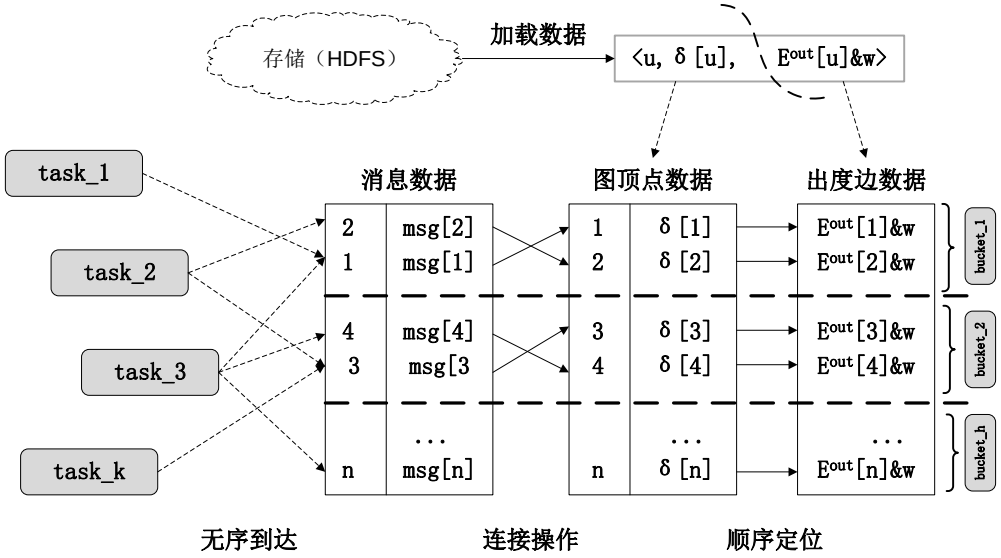


图 4.3 列存储模型

Fig. 4.3 The oriented-column storage model

静态哈希索引，是将同一个任务的接收的消息数据和图数据，按照图顶点的编号，进行取商运算，因此，可以保留同一个哈希桶内的数据局部性。其中，图顶点数据和出度边数据，在数据加载阶段，即按照指定的哈希函数，划分到不同的哈希桶中。而迭代过程中接收到的消息数据，则在接收过程中，按照相同的哈希函数，划分至不同的哈希桶。每个哈希桶，在内存中均有消息接收缓存，用于缓存该桶内的部分消息。当缓冲区溢出时，将缓冲区的数据保存为一个临时磁盘文件，然后清空缓冲区，用于接收新的消息数据。当完成消息接收操作，开始本地计算之前，隶属于同一个哈希桶的所有消息数据（包括磁盘上的临时文件）将执行一趟合并操作，使得同一个图顶点接收到的消息，最终被合并为一条消息。

如图 4.4 所示，每次迭代，均以哈希桶为单位完成本地计算任务。具体的，在第 *i* 步，对于第 j 个哈希桶，如果在第 *i*-1 步，第 *j* 个哈希桶的消息接收缓存溢出过，则其在磁盘上，将有若干消息临时文件（即 Message Spill File）。在处理第 *j* 个哈希桶之前，首先在内存开辟一段缓冲区用于执行消息合并。由于消息的可合并性，因此，只需要为每个图顶点分配一条消息的内存空间即可。所以缓冲区的大小，等于该桶内的图顶点数目与一条消息的字节规模的乘积。执行合并之后，该哈希桶收到的所有消息均驻留内存并以哈希方式组织。然后，从磁盘上逐条加载第 *j* 个哈希桶内的图顶点数据和出度边数据，根据图顶点 ID 完成消息数据和图顶点数据之前的连接操作，执行本地计算，然后更新图顶点数据的值，并按照用户定义的逻辑，按照出度边发送新的消息。需要发送消息，

按照目的顶点所在的任务为单位组织并异步发送[9]。当发送消息缓冲区溢出时，将阻塞本地计算过程，直到发送缓冲区的消息已经由发送线程推送至目的任务，而不会通过磁盘缓存。如果第 j 个哈希桶在第 i 步，没有接收到消息，则第 j 个哈希桶的本地图数据将会被跳过。因此，基于列存储的静态哈希索引方法，可以提高消息数据和本地图顶点数据连接操作的局部性，使每次迭代，本地图数据，至多与磁盘交互一次。

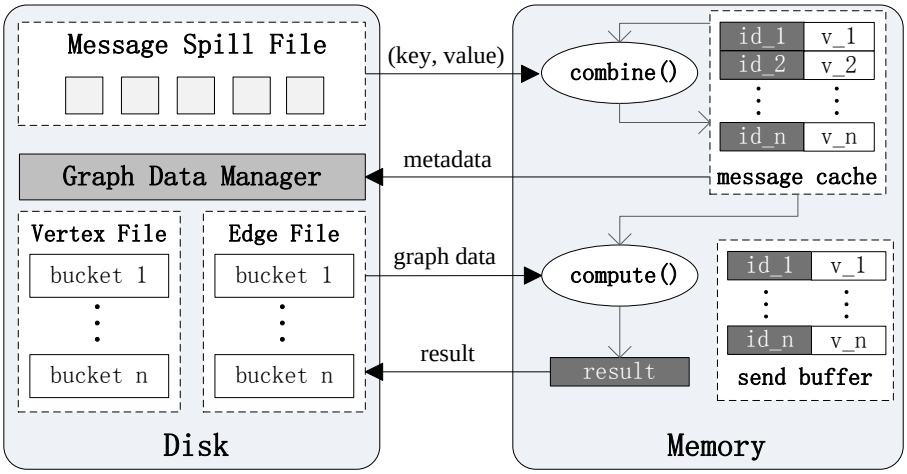


图 4.4 静态哈希索引处理流程

Fig. 4.4 Illustration of the static hash index

为提高并行处理效率，消息的组织处理和本地图数据的计算处理是异步并行执行的。如图 4.5 所示，在内存中缓冲多个合并后的哈希桶的消息数据（缓冲的哈希桶的个数，大于等于 2）。在本地图数据计算处理过程中，缓冲区内的消息数据不断被消耗，同时，消息预处理线程以哈希桶为单位，不断将合并后的消息放入缓冲区。当缓冲区溢出时（可能是计算处理线程执行速度较慢或者被消息发送线程阻塞），消息预处理过程被阻塞。

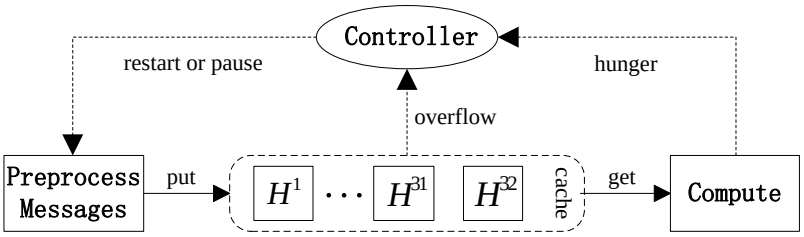


图 4.5 消息的 pipeline 处理过程

Fig. 4.5 Illustration of pipelining messages

静态哈希索引的一个关键问题是如何确定合适的哈希桶数目。采用静态哈希索引的主要目的，是提高连接操作的局部性，防止随机磁盘访问，因此，哈希桶数目的约束条件，即为内存的分配使用。由于内存缓冲区优先分配给消息，所以，消息缓冲区的管理，决定哈希桶的数目。通过上述分析（图 4.4 和图 4.5），内存主要分为三部分：消息发送缓存，消息接收缓存和消息处理缓存。为简化计算，我们用消息的数目表示内存的大小。

设 B_{task} 为一个任务的可用内存容量，V_{task} 为该任务负责处理的图顶点，则该任务的内存分配必须满足公式 4.1 的约束：

$$(N_t-1)B_s+2N_tN_bB_r+\frac{|V_{task}|}{N_b}L_c=B_{task} \tag{4.1}$$

其中，N_t 为该图处理作业的任务数目，而 B_s 是基本发送缓存单元（待发送的消息以目的顶点所在的任务为单位组织）。由于每个任务都可能向其它所有任务发送数据，因此，一个任务的消息发送缓存总大小为 $(N_t-1)B_s$。N_b 即为待确定的哈希桶数目。由于同一时刻，一个任务可能同时接收所有任务（包括自身）发送给同一个本地哈希桶的消息，而且各桶内的消息缓冲区中的消息，不断进行着合并操作，因此，需要对消息接收端的各桶的内存缓冲区进行加锁控制以保证正确性。由于消息收发的频率很高，频繁的加锁控制，将严重影响系统处理效率[9]。为消除加锁控制，消息接收端将每个本地哈希桶的消息接收缓冲区划分为 N_t 份，即为每个消息源（即发送消息的任务）在每个本地哈希桶的消息接收缓冲区均有一个消息接收缓冲单元，即为 B_r。在第 *i* 步中，对于尚未被处理的消息桶，其接收的第 *i*-1 步的消息，部分驻留内存缓冲区，而由于消息的异步发送，还可能接收来自第 i 步的消息，整两部分信息，应该同时分别存放，所以消息接收缓存的总规模为 $2N_tN_bB_r$。消息处理缓存是为消息数据的 pipeline 服务的（图 4.5），L_c 表示消息接收缓存中，缓冲的消息桶的数目，为实现并行处理，应有 $L_c>1$. 对于增量迭代计算，发送给同一个目的图顶点的消息，最终可以被合并为一条消息，因此，每个桶需要缓冲的消息的条数的上限，即为该桶内的图顶点数目。由于图顶点是数字连续编号，故本地哈希桶内的图顶点数目是平均分配的，所以消息处理缓冲区的规模为 $\frac{|V_{task}|}{N_b}L_c$。为保证 N_b 有解，根据公式 4.1 有：

$$B_r\le\frac{\left[B_{task}-(N_t-1)B_s\right]^2}{8N_tL_c|V_{task}|} \tag{4.2}$$

较大的 B_r 可以减少消息数据的磁盘存取开销，因此，在公式 4.2 中，等号成立的情况即为 B_r 的最优取值。将 B_r 的最优值带入公式 4.1 可得该任务的本地哈希桶的数目为：

$$N_b=\frac{B_{task}-(N_t-1)B_s}{4N_tB_r} \tag{4.3}$$

因此，只要设定了 B_s 的值，就可以计算得到 B_r（公式 4.2）和 N_b（公式 4.3），而 B_s 值，需要考虑网络的通信负载和通信速度等因素，受物理集群的硬件配置影响。

4.2.2 基于状态转换与 Markov 模型的动态 Hash 索引策略

由于出度边的规模不确定，而图顶点数据和对应的出度边数据，是同时加载的，因

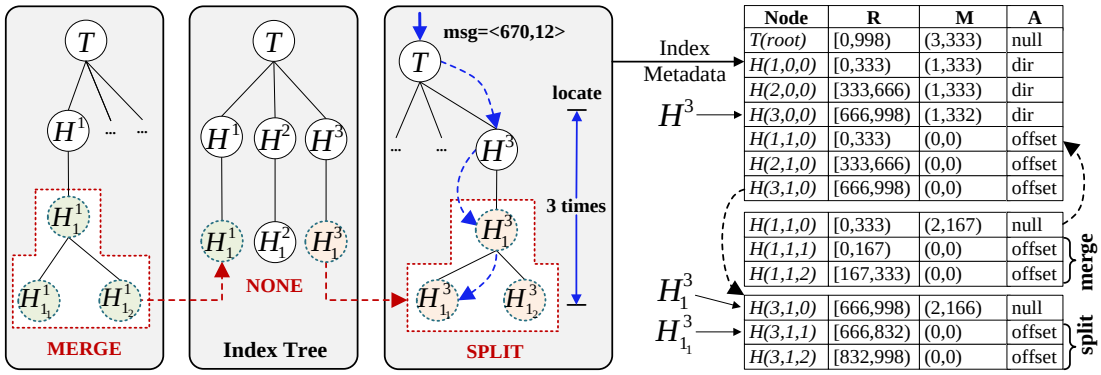
此，我们以图顶点数目作为图数据的磁盘存取开销的衡量标准。为便于分析，我们定义图数据加载比例 $LR = |V_l| / |V|$，而图数据加载效率 $LE = |V_p| / |V_l|$，其中，$|V_l|$表示从磁盘加载的图顶点数据的规模，而$|V_p|$表示被处理的图数据的规模。显然，较低的 *LR* 和较高的 *LE*，可以避免提取无效数据，提高磁盘存取的效率。静态哈希索引，虽然可以避免连接操作过程的随机磁盘读取，并在一定程度上，可以降低 *LR*，提高 *LE*（无消息的数据桶将被跳过），然而，根据状态转换模型，迭代过程中，$|V_p|$是动态变化的，因此，静态哈希索引的效果有限。为充分利用$|V_p|$的动态变化，我们提出了基于 Markov 模型的动态哈希索引策略，通过动态调整哈希桶的切分粒度，进一步降低 *LR*，提高 *LE*。

在动态哈希索引中，每个哈希桶的元数据是按照树结构组织的，如图 4.6 所示。在索引树中，共包含两个节点：（1）消息节点（如 H^1）；（2）数据节点（如 H_1^1）。消息节点，是管理消息接收缓存的基本管理单位，用于接收、合并、预处理消息。而数据节点，是管理本地图数据的基本单位，用于加载、计算处理、保存图数据。在任务初始化时，索引树中每个消息节点只有一个子节点，即数据节点。如第二个消息节点 H^2，其孩子节点为 H_1^2，即该数据节点是第二个消息节点在初次递归分割（即建立初始索引）的第 1 个叶子节点。消息节点数目的确定，与 4.2.1 小节中静态哈希索引索引的哈希桶数据的计算方法一致。考虑到消息数据操作的复杂性（接收合并、磁盘操作、预处理等），消息节点的数目是固定不变的，而数据节点，则可以被递归分割。在初始阶段，每个数据节点（即叶子节点）对应的图数据（图顶点数据和出度边数据）分别存放在不同的文件中（与 4.2.1 中的静态哈希索引相同）。而数据节点的递归分割，只是进行逻辑分割，并无物理操作。即，一个数据节点如果被分割成两个子节点，则其对应的磁盘文件并没有被切分，两个叶子节点分别记录对应的文件偏移量和长度，以节省递归切分的开销。

索引树中节点的元数据信息是一个三元组{*R, M, A*}，如图 4.6 所示，其中，*R* 代表该节点负责的图顶点的编号范围，*M* 是一个 *Key-Value* 对，*key* 是直接后继节点的数目，*value* 代表字节点的平均图顶点数目，计算方法为：$value = \lceil R.length / key \rceil$。*A* 是位置标示符，对于消息节点而言，*A* 是该哈希桶对应的消息数据在磁盘上的临时缓冲文件所在的目录；对于索引树的叶子节点（叶子节点一定是数据节点），则代表该哈希桶对应的图顶点数据和出度边数据的文件起始偏移量与长度；对于索引树中除消息节点之外的内部节点，*A* 值无意义。

隶属于同一个消息节点的叶子节点，在迭代过程中可以被递归的切分与归并。如果叶子节点 H_j^i 递归分割为 N_j^i 个子节点 $H_{j_k}^i$，其中，$1 \le k \le N_j^i$。在递归切分过程中，父节点的图顶点数据和出度边数据，将按照图顶点数目，平均切分到各叶子节点且各叶子节点内部的图顶点数据依然保持顺序编号。切分过程并不涉及具体的物理磁盘数据存取，

仅是元数据信息的更新，包括父节点 H_j^i 和新建叶子节点 $H_{j_k}^i$ 的元数据的维护。如图 4.6 中的 H_3^1，被分割为两个叶子节点 $H_{3_1}^1$ 和 $H_{3_2}^1$。则 H_3^1 的元数据信息 H(3,1,0)，内容由{[666, 998), (0, 0), offset}变为{[666, 998), (2, 166), null}，即被切分为 2 个叶子节点，每个叶子节点的最大图顶点数目为 166，由于 H_3^1 被切分后不再是叶子节点，因此 *A* 值为 null.而新增叶子节点叶子节点 $H_{3_1}^1$ 和 $H_{3_2}^1$ 的元数据分别为{[666, 832), (0, 0), offset}和{[832, 998), (0, 0), offset}，其包含的图顶点的数目均为 166。归并操作是切分操作的逆过程，仅发生在叶子节点的直接父节点（除消息节点），即同一个父节点的叶子节点才可以归并，不同父节点的叶子节点之间，互不干扰。叶子节点被归并后，原来的父节点就成为新的叶子节点。如图 4.6 中，当叶子节点 $H_{1_1}^1$ 和 $H_{1_2}^1$ 被归并后，父节点 H_1^1 成为新的叶子节点，对应的，元数据信息也要调整。



Node	R	M	A
T(root)	[0,998)	(3,333)	null
H(1,0,0)	[0,333)	(1,333)	dir
H(2,0,0)	[333,666)	(1,333)	dir
H(3,0,0)	[666,998)	(1,332)	dir
H(1,1,0)	[0,333)	(0,0)	offset
H(2,1,0)	[333,666)	(0,0)	offset
H(3,1,0)	[666,998)	(0,0)	offset
H(1,1,0)	[0,333)	(2,167)	null
H(1,1,1)	[0,167)	(0,0)	offset
H(1,1,2)	[167,333)	(0,0)	offset
H(3,1,0)	[666,998)	(2,166)	null
H(3,1,1)	[666,832)	(0,0)	offset
H(3,1,2)	[832,998)	(0,0)	offset

图 4.6 动态哈希索引的分割与合并

Fig. 4.6 Illustration of splitting and merging of the tunable hash index

递归切分的主要目的，是借助于哈希索引，避免提取无用的图数据。因此，在消息接收阶段，需要判定每条消息所归属的叶子节点。在本地计算阶段，如果某个叶子节点没有收到消息，则该桶内的图数据就被跳过。对于发送给顶点 *v* 的一条消息 *msg*[*v*]，其叶子节点的定位操作需要遍历索引树，在某个内部节点 H_j^i，定位该消息所述的叶子节点 $H_{j_k}^i$ 的公式如下：

$$k = \frac{getID(msg(v) - \theta)}{M.value} + 1 \tag{4.4}$$

其中，θ 为 H_j^i 的元数据 R 的最小顶点编号。在一次迭代中，假设收到的消息总量等于出度边数目，则消息定位操作的时间复杂度为 $\Omega((h-1)\cdot|E|/N_t)$，其中，$h$ 为索引树的高度，而 N_t 为任务数目。如图 4.6 所示，对于发往编号为 670 的图顶点的消息<670, 12>，需要通过 3 次定位操作，才可以找到其所属的叶子节点 $H_{3_1}^1$。

下面本文将介绍动态 Hash 索引的的调整类型判定方法和 Markov 评估模型。

（1）动态调整哈希索引

虽然切分叶子节点可以提高该叶子节点内图数据的 *LE*，但是切分的时机和切分的孩子节点的数目，是影响性能的两个关键因素。

算法 4.1（见表 4.1）用于获取第 t_{i+1} 步的叶子节点的全局调整类型，也即切分叶子节点的时机。该方法，同时也是一种自底向上的判定图状态的方法。所谓自底向上，即通过各个叶子节点的期望调整类型，决定全局调整类型，进而判定图的全局状态。调整类型 *AT* 包括三类：切分（Split），归并（Merge）和维持（None）。算法 4.1 是一个主从式运行结构，在每两个超级步之间的全局同步过程中调用。在第 t_i 步，首先，第 *k* 个任务，对其本地索引树的每个叶子节点，根据 LE_i^k 和 *K*，判定各个叶子节点的期望调整类型，并记录相关的统计信息（第 10-14 行）。$vector^k$ 是第 *k* 个任务的统计向量，共三个元素，分别表示期望被切分、归并和维持的叶子节点的数目。*type* 是一个数字标识，如果调整类型为切分，则 *type=0*，对于归并，*type=1*，维持，*type=2*。S_i^k 是第 *k* 个任务中，第 *i* 个叶子节点的统计信息。而 LE_i^k 则是第 *k* 个任务中，第 *i* 个叶子节点，在第 t_i 步的图数据加载效率。*K* 是关于激活图顶点变化规律的模拟曲线在某一阶段的斜率，表征了激活状态的图顶点规模的变化趋势。第 t_i 步的 *K* 值，实际是第 t_{i-1} 步的估算值。

各叶子节点判定完毕后，所有的统计信息被作业控制中心 JobInProgress 收集（第 2-4 行）。作业控制中心记录已完成的各步迭代中，激活状态的图顶点的规模，并根据最近的 K_Δ 步的值，估算局部拟合虚线的斜率 *K'*，该值将在第 t_{i+1} 步使用（第 5 行）。考虑到鲁棒性和准确性，本文一般取 $K_\Delta = 5$。全局调整类型的判定，是根据各任务的汇报结果，选择叶子节点数目最大的期望调整类型（第 6 行）。最后，作业控制中心将全局调整类型和第 t_i 步的斜率估算值发送给各个任务。在第 t_{i+1} 步，如果叶子节点的期望调整类型与 *AT* 匹配，则执行调整，否则维持不变。

图的迭代状态需要根据全局调整类型 *AT* 和第 t_{i+1} 步的斜率进行判定。判定规则如下：

（1）扩张态，$AT \in \{Split, None\} \& K^{'} > 0$；

（2）稳态，$AT \in \{Merge\}$；

（3）收缩态，$AT \in \{Split, None\} \& K^{'} \geq 0$；

（2）Markov 代价评估模型

在算法 4.1 中，各叶子节点通过调用函数 *getAdjustType(K,* LE_i^k*)*判定自己的期望调整类型。在判定函数中，首先评估该叶子节点的切分收益，如果收益为正，则返回值为 Split，否则返回 None。如果同一个父节点的所有叶子节点，其期望调整类型均为 None，则对该父节点调用 *getAdjustType* 函数，假设将该父亲节点切分的叶子节点的数目等于当

前叶子节点数目，评估收益，相似的，如果收益为负数，则所有叶子节点的调整类型为 Merge。

表 4.1 叶子节点的全局调整类型判定算法

Table 4.1 The algorithm of judging the global adjusting type for leaf nodes

算法 4.1 *adjustLeafNode(S, K)*

输入：当前超级步 t_i 的统计信息 S，第 t_{i-1} 步的斜率 K

输出：第 t_{i+1} 步的全局调整类型 AT，第 t_i 步的斜率 K'

1. **JobInProgress**
2. wait until all tasks report the $vector^k$ and $active^k$
3. $vector = \sum_{k=1}^{N_t} vector^k$ /* N_t: the number of tasks */
4. $active = \sum_{k=1}^{N_t} active^k$
5. put *active* into *HistoryQueue* and estimate K' by the last K_Δ
6. $AT = \max\{vector(i) | 0 \le i \le 2\}$
7. send {*AT, K'*} to each task
8. **Task k**
9. $vector^k$ = <0, 0, 0>
10. **while** $S^k \neq \varnothing$ **do**
11. S_i^k = remove one from S^k
12. $type = getAdjustType(K, LE_i^k)$
13. $vector^k$[type]++;
14. $active^k = active^k + getActive(S_i^k)$
15. **end while**
16. send $vector^k$ and $active^k$ to **JobInProgress**
17. **wait until JobInprogress returns {*AT, K'*} and then set *K* = *K'***

切分操作的收益，依赖于切分的子节点的数目。本文采用 Markov 模型来评估切分的收益，确定最优切分值。对于叶子节点 H_j^i，假设将其切分为 N_j^i 个孩子节点，其中，第 k 个孩子节点 $H_{j_k}^i$ 的图顶点集合为 $V_{j_k}^i$，而 ${}^tV_p^{ij}$ 为第 t 个超级步中，哈希桶 H_j^i 中被处理的图顶点集合，则有 ${}^tV_p^{ij} \subseteq (\cup V_{j_k}^i) = V_j^i$，其中 V_j^i 为 H_j^i 中的图顶点集合。${}^t\Lambda$ 为第 t 步中，接收到消息的叶子节点的集合，则有：${}^t\Lambda = \{k | {}^tV_p^{ij} \cap V_{j_k}^i \neq \varnothing, 1 \le k \le N_j^i\}$。

定理 4.1 对于哈希桶 N_j^i，假设随机变量 $X(t)$ 为第 t 步收到消息的叶子节点的数目，即 $|{}^t\Lambda|$，则随即处理过程 $\{X(t), t \in T\}$ 是一个齐次 Markov 链，其中 T 为超级步计数器集合，$T = \{0, 1, 2, ..., t_{up}\}$，而 t_{up} 是随机处理过程的转换过程的上限。

证明：在增量迭代处理过程中，Markov 转换过程的时间点集合，可以视为超级步计数器集合，而状态空间集合为 $I = \{a_i | 0 \le a_i \le N_j^i\}$，即 $|{}^t\Lambda|$。根据定义，${}^t\Lambda$ 表征了第 t 步，接收的消息数据在 N_j^i 的孩子节点之间的分布。在第 t 步，图顶点根据其第 t-1 步的值和收到的消息值，进行本地计算并发送新的消息。因此，${}^{t+1}\Lambda$，${}^{t+2}\Lambda$，…，${}^{t+n}\Lambda$ 仅依赖

于 $^{t}\Lambda$。而从 $^{t}\Lambda$ 到 $^{t+1}\Lambda$ 的转移概率由 $^{t}\Lambda$ 和图顶点本身的值 δ 决定。所以 X(t)具有 Markov 属性。由于 I 和 T 均是独立的，所以 $\{X(t), t \in T\}$ 是一个 Markov 链。进一步的，在概率转移矩阵 P 中，有 $P_{xy}(t, t+\Delta t) = P_{xy}(\Delta t)$，因此 $\{X(t), t \in T\}$ 是一个齐次 Markov 链。

由定理 4.1 可知，原始的转移概率，可以通过采样进行估算。在 t_m 步，可以从 $^{t_m}V_p^{ij}$ 中抽取一个随机样本，然后计算消息数据在 N_j^i 个孩子节点之间的分布。为计算简单，我们假设从状态 a_x 到状态 a_y 之间的概率分布是一个等差数列，其公差 $d = (LE_i) \cdot K$，而最小值为 $^{(2y)}/_{x(x+1)}$。则一步转移概率 p_{xy} 可以被计算出来。而 Δm 步的概率转移满足 Chapman-Kolmogorov 等式。因此有公式 4.5：

$$P\{X(t_{m+\Delta m}) = a_y \mid X(t_m = a_x)\} = p_x(t_m) P_{xy}(\Delta m) \tag{4.5}$$

则在第 $t_{m+\Delta m}$ 步，被跳过的哈希桶的数学期望由公式 4.6 计算得到：

$$\Phi(N_j^i, t_{m+\Delta m}) = \sum_{x=1}^{N_j^i} \sum_{y=1}^{N_j^i} (N_j^i - x) p_x(t_m) P_{xy}(\Delta m) \tag{4.6}$$

考虑到 4.1 小节中关于消息定位的时间复杂度，我们可以使用公式 4.7 推断出切分的开销为：

$$\Psi(N_j^i, \Delta t) = \sum_{k=1}^{\Delta t} (T_{cpu} \cdot \Omega(\frac{\Delta h \mid E \mid}{N_t})) \tag{4.7}$$

这里，Δh 是完成切分操作后索引树的高度变化。特别的，如果 N_j^i 等于 1，则该表明哈希桶 H_j^i 不会被分割，索引树的高度不会发生变化，$\Delta h = 0$。T_{cpu} 则是执行一条指令的 CPU 时间开销。$\Delta t = t_{up} - t_m$。如果 $K<0$,则 tup 为用户定义的最大迭代步数，否则 $\Delta t = K_{\Delta}$。切分操作的收益，即部分子节点的图数据将不会被访问，所以切分 H_j^i 的收益为可由公式 4.8 计算得到：

$$\Psi'(N_j^i, \Delta t) = (\frac{byte(V_j^i) + byte(E_j^i)}{s_d N_j^i}) \sum_{\Delta m=1}^{\Delta t} \Phi(N_j^i, t_{m+\Delta m}) \tag{4.8}$$

其中，$byte(V_j^i)$ 为哈希桶 H_j^i 内的图顶点数据的字节规模，而 $byte(E_j^i)$ 则为出度边的存储开销，s_d 是磁盘访问的速率。切分数目 N_j^i 的候选值集合为：$C = \{< n, \rho > \mid 1 \le n \le \varepsilon\}$。其中 $\rho = \Psi'(n, \Delta t) - \Psi(n, \Delta t)$，$\varepsilon$ 用于控制索引树的存储规模，防止索引树超出内存容量。对于切分操作，我们可以按照如下方式寻找到最优切分数目 γ：在集合 C 中，选取 ρ 最大的元组 $< n, \rho >$ 构成子集 C'；$\forall < n, \rho > \in C'$，$\gamma$ 是其中最小 n 值。如果 $\gamma = 1$，则该节点的调整类型为 None，即不需要切分，否则，为 Split。

对于归并操作，我们可以将叶子节点的直接父节点视为叶子节点并假设 γ 为该父节点的孩子节点的数目，如果 $\rho < 0$，则该父节点的所有叶子节点的调整类型为 Merge。

4.3 实验结果与分析

4.3.1 实验设置

实验集群由 21 台计算机组成，1 台主控节点，20 台计算节点，每台计算机的配置为：2 Intel Core i3-2100 CPUs，8GB 内存和 500GB 硬盘，磁盘转速为 7200RMP。所有计算机安装 Linux RedHat 6.0 操作系统并由 1 台 Gbps 带宽的交换机互联。实验过程中，为避免同一台计算机上不同任务之间的内存与磁盘争用影响，每个计算节点仅运行一个任务，每个任务的 JVM 配置 1GB 内存。

实验采用单源最短路径计算作为示例程序。实验数据集如表 4.2 所示，其中 USA-RN 是全美路网图[40]。每个图顶点的初始值（最短路径值）为无穷大，无权图的权重设为随机数。对于 S-LJ 和 Wiki-PP 数据集，算法迭代至收敛，对于 USA-RN 数据集，由于图的直径较大且十分稀疏，收敛缓慢，故仅截取前 300 步的迭代统计信息。

表 4.2 实验数据集说明

Table 4.2 Characteristics of data sets

数据集	顶点	出度边	平均出度	文件大小
S-LJ	4,847,571	68,993,773	14.23	576MB
USA-RN	23,947,347	5,833,333	0.244	1.20GB
Wiki-PP	5,716,808	130,160,393	22.77	1.05GB

4.3.2 索引性能评估

本组实验用于对比静态 Hash 索引和动态可调 Hash 索引的性能。我们首先分析了当前集群配置下，消息发送阈值 B_s 的取值，然后对整个迭代过程中的综合性能、平均数据加载比例、顶点加载与处理规模等整体指标进行了对比分析，最后分析了不同数据集的每个超级步的性能和顶点加载与处理规模。由于动态可调 Hash 索引是针对本地图数据（图顶点及其出度边）的优化策略，因此本组实验中，我们将所有图数据保存至本地磁盘以验证其有效性。

消息发送阈值 B_s 测试。由 4.2.1 小节的分析可知，B_s 的大小将影响和 B_r 的设置，而 B_s 值受物理集群的网络通信、磁盘存取速度等硬件条件影响。我们在 S-LJ 数据集上运行单源最短路径算法，通过设置不同的 B_s 输入以确定最优取值区间。由图 4.7 可知，在本文的集群环境中，B_s 的最优取值区间为 3500 至 6200，即缓存 3500 至 6200 条消息。

整体性能分析。通过设置，如图 4.8 所示，基于 Markov 模型的动态可调 Hash 索引可以提高整体迭代处理效率，但是具体效果与数据集特点有关。

通过图 4.9 可知，对于 S-LJ 而言，动态 Hash 索引对于加载比例 *LR* 的优化效果有限，仅有减少了约 18.75%，而 USA-RN 和 Wiki-PP 则分别减少了 86%和 80.6%，即大

量的无效数据避免被反复存取。图 4.10 进一步展示了两种方法对于数据加载效率 *LE* 的影响。由于$|V_p|$为固定值，故 *LE* 与$|V_l|$成反比。对于 S-LJ，通过动态 Hash 索引，*LE* 由 20.9%提高到 25.8%，而 USA-RN 和 WikiPP 则分别由 2.24%和 3.38%提高到 15.3%和 17.4%。因此，图 4.8 中，对于 S-LJ，动态索引的运行效率是静态索引的 1.08 倍，效果甚微，而对于 Wiki-PP，则可以达到 2.02 倍，效果明显。值得注意的是，虽然 USA-RN 的 *LR* 和 *LE* 改善效果比 Wiki-PP 更加显著，但是效率提升比例仅有 1.37 倍，低于 Wiki-PP。这是因为 USA-RN 为稀疏图（平均出度仅 0.244），磁盘存取的数据规模极小，而迭代的 Warm-UP 开销（如磁盘与通信的初始化操作等）占据了总运行时间的绝大部分比例，所以，针对磁盘存取的数据规模进行优化的动态 Hash 索引，对于整体性能的提升，并不明显。

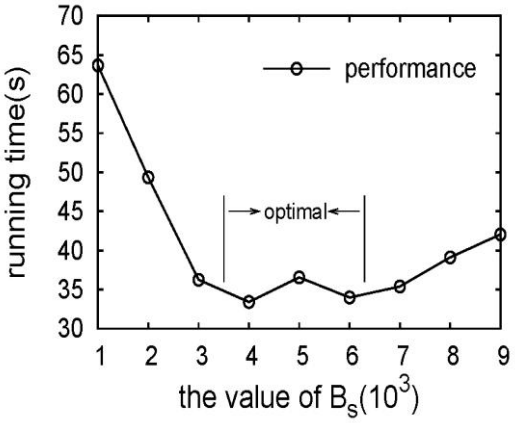


图 4.7 B_s 对整体性能的影响

Fig. 4.7 Influence of B_s

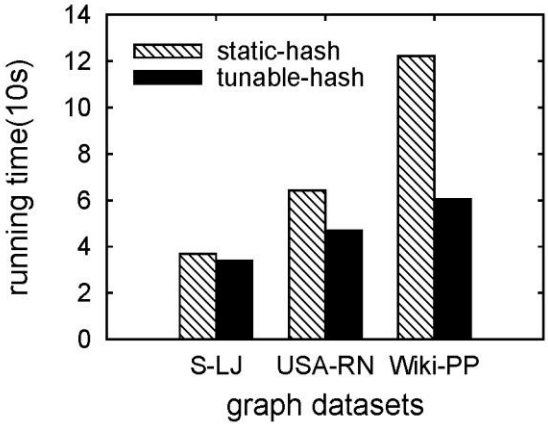


图 4.8 整体性能测试

Fig. 4.8 Overall performance

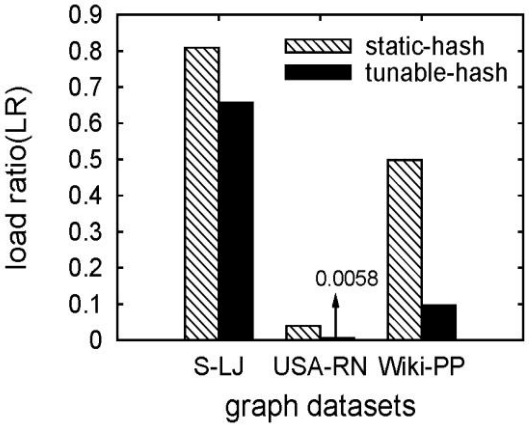


图 4.9 平均数据加载比例 *LR* 测试

Fig. 4.9 The average of *LR*

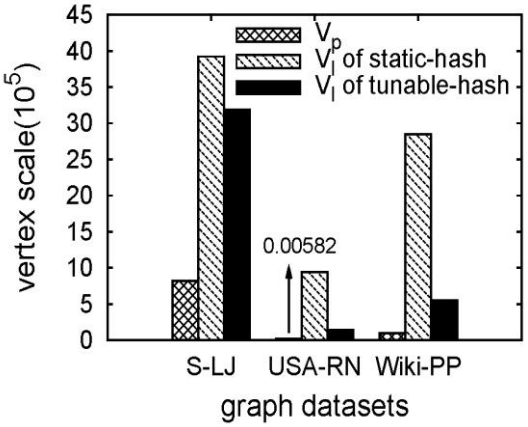


图 4.10 数据加载效率 *LE*

Fig. 4.10 The average of *LE*

迭代分析。图 4.11、图 4.12 和图 4.13 分别展示了 S-LJ、USA-RN 和 Wiki-PP 的各步迭代性能与数据加载效率 LE。从图 4.11(a)可知，S-LJ 在扩张态和收敛态的驻留时间较短，而动态 Hash 索引的“自底向上”侦测、调整机制，具有“滞后性”（如图 4.11(b)所示），导致动态 Hash 索引无法充分发挥作用，所以 S-LJ 的 *LE* 改善幅度较低，整体加

速比较低。反之，通过图 4.12(a)和图 4.13(a)可知，USA-RN 的扩张态和 Wiki-PP 的收敛态的驻留时间较长，因此动态 Hash 索引能够充分改善 *LE*（如图 4.12(b)和图 4.13(b)所示）。

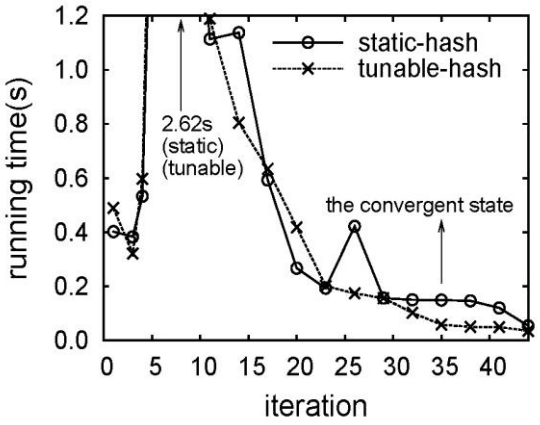


图 4.11(a) S-LJ 迭代性能

Fig. 4.11(a) The running time per iteration of S-LJ

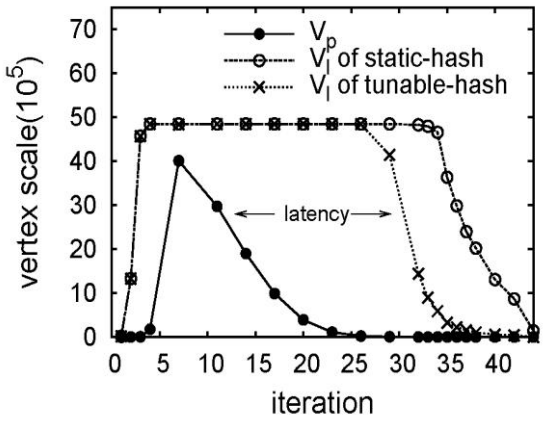


图 4.11(b) S-LJ 数据加载效率 *LE*

Fig. 4.11(b) The *LE* per iteration of S-LJ

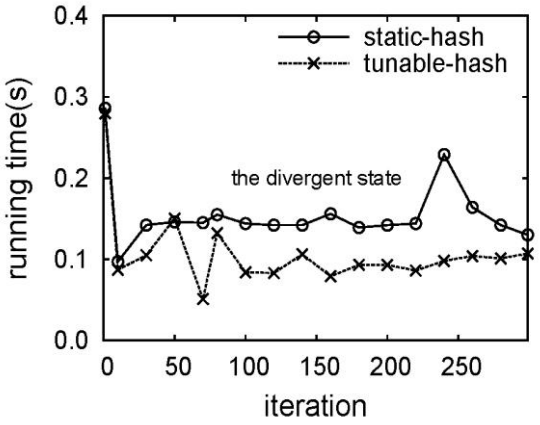


图 4.12(a) USA-RN 迭代性能

Fig. 4.12(a) The running time per iteration of USA-RN

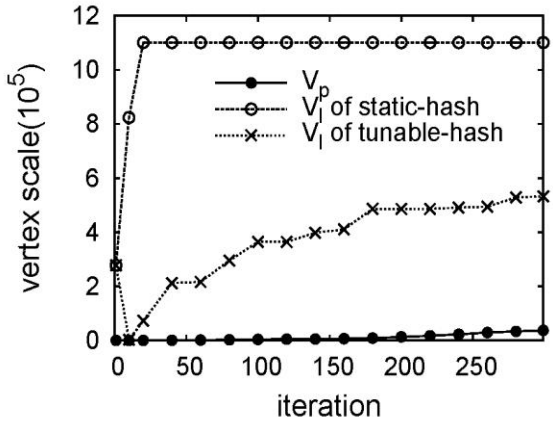


图 4.12(b) USA-RN 数据加载效率 *LE*

Fig. 4.12(b) The *LE* per iteration of USA-RN

4.3.3 数据处理能力与处理效率评估

DiterGraph 是本文开发的一个图处理原型系统，支持磁盘操作和动态可调 Hash 索引，详细介绍见第 6 章。本组实验，DiterGraph 仅开启磁盘和可调索引功能，仍采用单源最短路径算法作为示例程序，迭代至算法收敛。

图 4.14 对比了 DiterGraph、Giraph、Hama 和 Hadoop 的数据处理能力和处理效率。实验采用 10 个计算节点，启动 10 个任务，每个任务的内存为 500MB。输入数据集为模拟数据集，顶点规模从 100 万至 500 万，平均出度为 13.5。从图中可知，DiterGraph 的处理性能是 Giraph 的 2 倍，是 Hadoop 的 17 倍，对于 Hama 可以达到 43 倍。进一步的，由于 Giraph 假设数据全部驻留内存，当顶点规模达到 400 万时，Giraph 内存溢出，无法正常运行。因此，借助于磁盘管理和可调 Hash 索引，DiterGraph 能够利用有限的

资源，高效的处理大规模图。

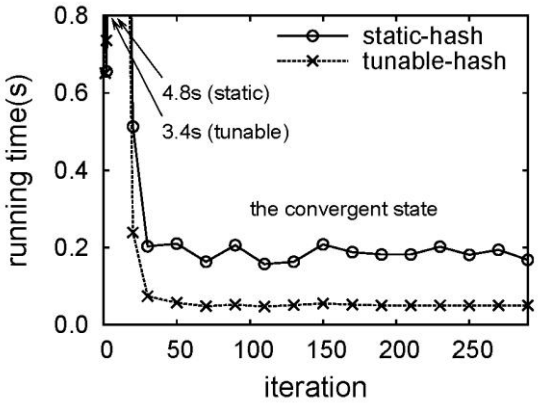


图 4.13(a) Wiki-PP 迭代性能

Fig. 4.13(a) The running time per iteration of Wiki-PP

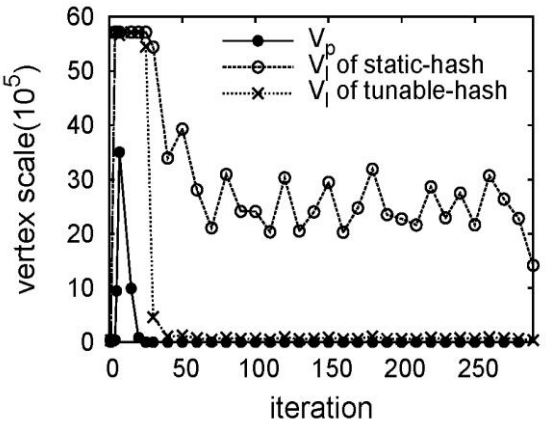


图 4.13(b) Wiki-PP 数据加载效率 *LE*

Fig. 4.13(b) The *LE* per iteration of Wiki-PP

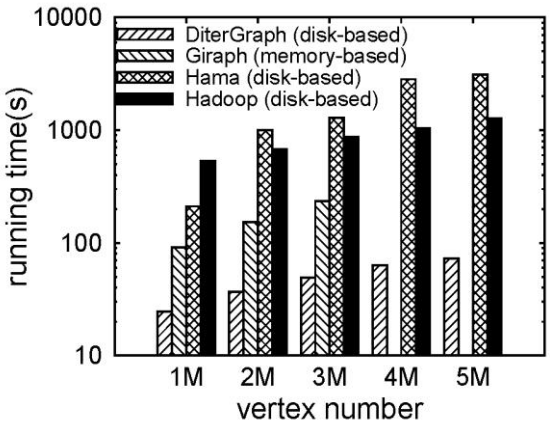


图 4.14 DiterGraph vs. 其它系统

Fig. 4.14 DiterGraph vs. Other systems

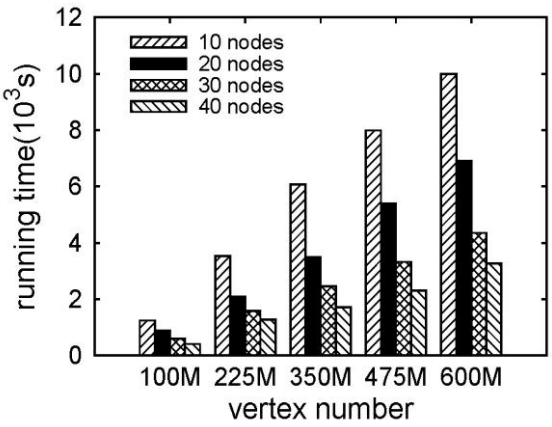


图 4.15 DiterGraph 的可扩展性分析

Fig. 4.15 The scalability of DiterGraph

图 4.15 测试了 DiterGraph 的可扩展性。实验采用模拟数据集，平均出度 13.5。最大数据集拥有 6 亿顶点，85 亿条边，文件大小达到 114GB，最大计算节点数目为 40，每个节点启动一个任务，每个任务配置 2GB 内存。由图可知，当计算节点数目固定为 40，图顶点规模从 1 亿增加到 6 亿时，处理时间从 415 秒增加到 3262 秒，与图顶点规模几乎成先行增长。反之，给定输入图的规模为 6 亿顶点，当计算节点数目从 10 增加到 40 时，处理时间由 9998 秒降低到 3262 秒。

4.4 本章小结

本章在累加迭代的基础上，针对图应用引入了增量迭代的概念。通过分析典型算法的增量迭代计算过程，总结出增量迭代的三个特点：（1）消息的非关联完整性；（2）消息的可合并性；（3）激活状态图顶点规模的动态变化性。根据后两者，本章分析了图迭代过程中的状态转换关系：扩张态，稳态，收缩态。本章设计了基于列存储模型的静态哈希索引，提高了消息数据与图顶点数据之间连接操作的局部性，避免了磁盘随机访问，

减少了无效数据的存取开销。在静态哈希索引基础上，基于 Markov 模型的动态哈希索引，能够充分利用图顶点规模在迭代过程中的动态变化，进一步避免读取无效的图数据，提高整体处理效率。

第5章 增量迭代的消息优化技术

关于消息通信的优化处理，Pregel、Giraph 等图处理系统以及 Hadoop 等通用云计算系统，都有相关工作，处理的思路也基本一致，即根据具体应用，定义 combine 操作，在发送端和接收端对消息进行合并处理。在继承现有消息优化机制的基础上，本章将深入探究分析增量迭代处理中的消息特点，然后结合 BSP 模型对其进行分类优化处理。

5.1 基于 EBSP 模型的 Hybrid 迭代机制

5.1.1 同步与异步迭代机制分析

对于消息具有关联完整性约束限制的应用，如经典 PageRank 算法，如果顶点 v 没有收到所有的入度边信息即启动本地计算，会导致最终的计算结果不可用。在这种情况下，BSP 模型的超级步和全局同步，实际上是提供了一种保障机制：确保所有的图顶点在每次迭代过程中，都已经收到了所有消息。由于增量迭代图算法的消息无关联完整性（见 4.1.2 小节）约束，因此可以采用同步机制或异步机制实现迭代计算[5,6]。

增量迭代的实现，是每个图顶点迭代执行 $v^k = v^{k-1} \oplus \Delta v^k$ 操作的过程，通过 Δv^k 不断修正 v^k，使其不断逼近稳态值。增量迭代的 BSP 同步实现，在每次迭代中，每个顶点收到全部理论消息值（对于单源最短路径和增量 PageRank 计算等，由于收敛性，收到的消息条数的上限为其入度值）后，才执行本地计算。例如，在第 i 步迭代过程中，顶点 v 共收到两条消息 $msg_i^{t_1}[v]$ 和 $msg_i^{t_2}[v]$，两者分别在 t_1 和 t_2 时刻到达且 $t_1 < t_2$。在同步机制下，这两条消息将会执行 $\oplus$ 操作，得到合并后的 Δv^k，然后读取本地图数据，执行 $v^k = v^{k-1} \oplus \Delta v^k$ 并发送新消息。因此，每个图顶点在每次迭代中，至多进行一次本地计算，发送一次消息。这有两个优势：一方面，在图数据驻留磁盘的情况下，BSP 同步可以减少本次磁盘存取开销；另一方面，每步迭代的消息通信规模可以得到控制。但是，同步迭代，必须保证每个图顶点收到全部理论消息后，才可以执行下一步迭代，对于单个图顶点而言，消息的传播因同步等待而有所延迟。

对于异步迭代实现，各顶点独自处理消息，与其它顶点之间不存在同步约束，从而加快了消息的传播速度，提高收敛性能。在极端情况下，异步迭代过程中，图顶点每接收一条消息即可以启动一次计算并发送新的消息，这会导致频繁的顶点更新并增大消息的发送量。为便于和 BSP 同步比较，假设在 BSP 个一次全局同步过程中，异步迭代实现时，第 i 个图顶点将被更新 α_i 次，而更新一次，发送的消息数目的上限为 β_i，由于异步处理过程中，消息数据可以被及时处理，此处假设异步迭代的消息数据不会驻留磁盘，则一次“同步”过程中，异步迭代的消息通信和图数据读取总开销由公式 5.1 得到：

$$\frac{1}{|P|}\times C_{communicate}\times\eta+\frac{1}{|P|}\times C_{disk}\times\omega \tag{5.1}$$

其中，|P|为任务数目，$C_{communicate}$ 为一条消息的通信开销，而 C_{disk} 为一个图顶点对应的图数据的本地磁盘存取开销。η 为总消息规模，且 $\eta=\sum_{i=1}^{|V|}\alpha_i\beta_i$，而 ω 为图顶点数据的读取次数，$\omega=\sum_{i=1}^{|V|}\alpha_i$。此外，异步实现过程中，由于消息到达的无序性和较差的局部性，将导致大量的磁盘随机存取开销，即使采用动态哈希索引策略，这种开销也十分巨大。在 BSP 同步迭代过程中，η 将约减至 $\sum_{i=1}^{|V|}\beta_i$，而 $\omega=|V|$。当然，由于同步处理过程中，消息不能即到即处理，可能需要缓存。假设一个超级步的所有消息均在接收端缓存磁盘，其存储规模与出度边相当。而异步处理过程中，$\alpha_i\geq 2$（如果 $\alpha_i=1$，将与同步处理类似，所以不考虑这种情况），故消息的存储开销，被本地图数据的频繁更新所抵消。

因此，在增量迭代的实现方式中，同步实现和异步实现，各有利弊。而然，如果图数据驻留磁盘，则异步实现，将导致致命的磁盘 I/O 开销。此外，异步实现在消息通信方面的开销，远高于同步实现，尤其是高入度的稠密图。

5.1.2 典型算法分析

根据 5.1.1 小节的分析，对于磁盘驻留的大图迭代，应选择 BSP 同步迭代实现方式。虽然同步迭代，可以约减消息规模，但是顶点之间的同步等待，延迟了消息的传播速度，进而影响收敛效率。本节将以单源最短路径计算和 PageRank 计算为例，分析 BSP 同步实现方式的弊端。

如图 5.1 所示的单源最短路径计算过程。在具体分析之前，首先定义多路径顶点（定义 5.1）和冗余消息（定义 5.2）概念。

定义 5.1 多路径顶点（Multipath-Vertex）。给定一个有向图，从源顶点 v_s 开始，如果存在一条路径 P_{st}，使得从 v_s 开始，经过 P_{st} 可以到达顶点 v_t，且路径 P_{st} 经过的图顶点数目为 i+1，则称顶点 v_t 为第 i-hop 可达顶点，而 V_i 为所有 i-hop 可达顶点的集合。集合 V_{mul} 定义为：$V_{mul}=\{v\,|\,v\in V_i\wedge v\in V_j\wedge\ldots\wedge v\in V_k, i\neq j\neq k\}$。则 $\forall v\in V_{mul}$，称为多路径顶点。

定义 5.2 冗余消息。假设顶点 v 是一个多路径顶点且 $v\in V_i\wedge\ldots\wedge v\in V_j\wedge\ldots\wedge v\in V_k, i<j<k$，顶点 v 在不同迭代步数中收到的消息经过合并后，分别为 $msg_i^{\Delta}[v]$, …, $msg_j^{\Delta}[v]$, …, $msg_k^{\Delta}[v]$. 如果 $msg_i^{\Delta}[v]>\ldots>msg_j^{\Delta}[v]>\ldots msg_k^{\Delta}[v]$，则称第 k 步之前收到的消息均为冗余消息。

冗余消息的存在，导致大量的消息通信，同时，也会导致本地图数据被无效存取、

处理。

BSP 同步迭代实现，本质是广度优先搜索过程。每迭代一次，消息按照图的拓扑结构传播一跳。多路径顶点处于不同路径的不同跳数，因此，在迭代过程中，会被处理多次。特别的，如果顶点 v 属于 k 个不同的可达顶点集合，则 v 至少会被处理 k 次。多路径顶点具有传播性，即 v 为多路径顶点，则 v 的后继顶点均为多路径顶点。在 5.1.1 小节中已经说明 BSP 的同步迭代实现，可以约减冗余消息规模，进而减少无效的磁盘存取开销。然而，进一步分析可以发现，多路径顶点的存在，仍然会导致冗余消息的生成和无效磁盘存取。

如图 5.1 所示，s 为单源最短路径计算的源顶点，则 1-hop 顶点集合为 V_1 = {a, b, c, d, e}, 2-hop 顶点集合为 V_2 = {e, f, g}. 显然 $e \in V_1 \bigcap V_2$，e 是一个多路径顶点。根据多路径顶点的传播性，顶点 f, g, h 和 i，均为多路径顶点。在 BSP 同步迭代的第一步，顶点 e 的值为 4，然后 e 发送消息给 f 和 g。在第 2 步，e 又收到来自顶点 a 和顶点 d 消息，分别为 2 和 3。其中，2 与 3 被合并为一条消息，值为 2，并与 e 本身的值 4 对比，所以 e 的最短路径值更新为 2，并再次向 f 和 g 发送新的消息。而顶点 e 在第一步发送的消息，成为无用消息。因此，顶点 f, g, h, i 均会被更新两次。

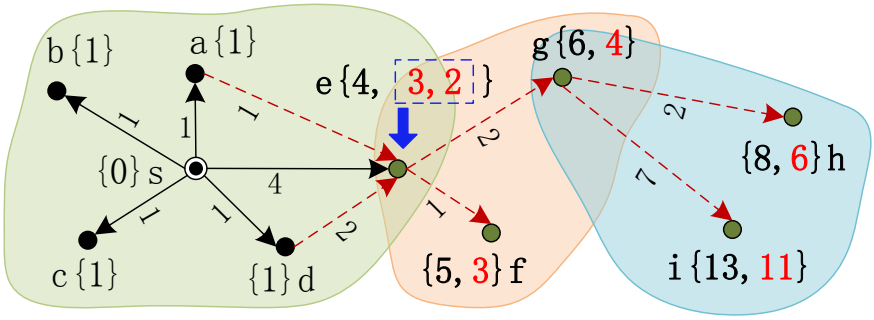


图 5.1 单源最短路径计算的 BSP 同步实现

Fig. 5.1 BSP-based implementation of the single source shortest path computation

连通域计算，也是不断广播顶点最小值的过程，其⊕操作就是取最小值，与单源最短路径相似，因此也存在冗余消息的现象。对于 PageRank 计算，⊕操作为求和运算，不会存在冗余消息的现象，但是，全局同步会延迟最新 PageRank 值的传递，影响收敛速度。

5.1.3 基于 EBSP 模型的 Hybrid 迭代机制

通过 5.1.1 小节的分析可知，在图数据驻留磁盘时，BSP 同步迭代的效率高于异步迭代。但是，同步迭代没有充分利用增量迭代对于消息无关联完整性约束这一特点，因此本节将在 BSP 模型的基础上，介绍扩展的 BSP 模型，EBSP（见定义 5.3）。

定义 5.3 EBSP 模型（Extended BSP Model）。设 M_{i+1} 为第 t_{i+1} 步处理的消息集合，

在第 t_i 步，由于消息发送和本地计算是异步进行的，部分第 t_{i+1} 步处理的消息已经被接收，即 $M_{i+1}^s \neq \varnothing, M_{i+1}^s \subseteq M_{i+1}$。则在第 t_i 步处理图顶点 v 时，$\forall m \in (M_{i+1}^s \bigcup M_i)$，顶点 v 均可以处理，但顶点 v 在第 t_i 步仅被处理一次。

BSP 模型通过严格的全局同步，将图顶点的更新和消息加以分离，不同超级步之间的消息是不可见的。而 EBSP 模型方框了对于消息的限制。EBSP 模型保留了 BSP 模型的同步更新特点，以减少磁盘存取开销和消息规模，但是连续两步之间的消息是可见的。

在 EBSP 模型下，增量迭代的 Hybrid 迭代计算模型为：

$$\begin{cases} \Delta v_{mem}^{k+1} = \sum\limits_{v_i \in \left[E_k^{in}[v]\right]_{mem}} \oplus(\Delta v_i^k) \\ v^k = v^{k-1} \oplus (\Delta v^k \otimes \Delta v_{mem}^{k+1}) \\ \Delta v^{k+1} = \sum\limits_{v_i \in E_k^{in}[v]} \oplus(\Delta v_i^k) \end{cases}$$

在第 k 步迭代，执行顶点 v 的计算更新时，需要综合考虑第 k 步收到的完整消息 Δv^k 和部分第 k+1 步处理的消息 Δv_{mem}^{k+1}。其中，第 k+1 的消息，仅计算当前驻留内存的部分消息，考虑到磁盘存取开销，忽略已经驻留磁盘的消息。另外，由于消息的异步发送且第 k 步迭代尚未进行全局同步，因此发给图顶点 v 且用于第 k+1 步的消息，是其第 k+1 步收到的全部消息的子集。Δv^k 和 Δv_{mem}^{k+1} 完成⊗运算后的结果，用于顶点 v 的本地计算。⊗运算包括两种类型的操作：

（1）剪枝运算，运算结果为布尔变量，如果返回值为 true，则执行正常运算，即 $v^k = v^{k-1} \oplus \Delta v^k$，如果为 false，则顶点 v 在本地迭代计算过程中，不执行本地计算处理，也不会发送消息。在剪枝运算中，第 k+1 步的消息子集是只读的，仅用于剪枝第 k 步收到的消息，以减少无效的磁盘存取开销和冗余消息规模，消息子集在第 k+1 步将会被正常处理。

（2）⊕运算，即⊗运算与⊕运算相同。此时，第 k+1 步的消息子集将从 M_{i+1} 中移除，直接参与第 k 步的本地计算处理，这可以加快消息的传播速度，提高收敛效率。

5.1.4 消息优化示例分析

本节将以单源最短路径计算和 PageRank 计算为例，介绍⊗运算的两种操作优化：跨步消息剪枝处理和跨步消息合并处理。用户可以根据特定的应用，灵活选择不同的优化操作。

（1）跨步消息剪枝处理（ASMP）

以单源最短路径计算为例，跨步消息剪枝处理，即⊗运算为剪枝运算，与基于 Markov 模型的动态哈希索引结合，减少迭代过程中冗余消息的生成，提高磁盘存取效

率。算法 5.1（见表 5.1）展示了对于消息节点 H^k 的剪枝操作处理流程。首先，按照 4.2.1 小节的处理，该桶内所收到的第 *t* 步的消息将在内存中完成合并操作，结果保留在 M_t^k。第 *t*+1 步的驻留内存的消息经过合并后，保存至 $M_{t+1}^{k_s}$。如果发送给顶点 u 的消息在 M_t^k 和 $M_{t+1}^{k_s}$ 中均存在且第 *t* 步的消息值大于第 t+1 步的消息值，则第 *t* 步的消息放入剪枝集合。经过剪枝操作后的第 *t* 步的消息 $M_t^{k_p}$，作为消息输入源，驱动对应的图数据的计算处理。特别的，第 *t* 步的消息经过剪枝操作后，会有更多的叶子节点无接收消息，则对应的图数据可以避免被计算处理。已有的消息合并技术，只是用于减少已生成消息的维护和网络通信开销，而 ASMP 方法，是利用下一步的消息剪枝本步消息，直接避免生成冗余的消息，二者并不冲突，可以联合使用，进一步提高系统的处理效率。

表 5.1 跨步消息剪枝算法

Table 5.1 The algorithm of ASMP

算法 5.1 *doASMP(*M_t^k, M_{t+1}^k*)*

输入：第 k 步接收的消息集合 M_t^k，第 k+1 步接收的消息集合 M_{t+1}^k

输出：剪枝后的第 k 步消息集合 $M_t^{k_p}$

1. set V_t^k = extract vertex IDs from M_t^k
2. set $V_{t+1}^{k_s}$ = extract vertex IDs from $M_{t+1}^{k_s}$
3. **foreach** $u \in V_t^k \bigcap V_{t+1}^{k_s}$ **do**
4. $msg_t^k[u]$ = *getMsg(*M_t^k, u*)*
5. $msg_{t+1}^{k_s}[u]$ = *getMsg(*$M_{t+1}^{k_s}$, u)
6. **if** $msg_t^k[u] > msg_{t+1}^{k_s}[u]$ **then**
7. put $msg_t^k[u]$ into the Pruning Set M_p
8. **end if**
9. **end foreach**
10. return $M_t^{k_p} = M_t^k - M_P$

对于单源最短路径计算，ASMP 方法对于冗余消息的剪枝效果可以由定理 5.1 估算得到。

定理 5.1 对于图顶点 v_r，如果 $\delta[v_r] > msg_t^k[v_r] > msg_{t+1}^{k_s}[v_r]$，则最大消息剪枝收益为 $\Gamma(v_r)$，

$$\Gamma(v_r) = \begin{cases} |E^{out}[v_r]|, v_r = v_{\max} \\ |E^{out}[v_r]| + \sum_{\forall v_m \in E^{out}[v_r]} \Gamma(v_m), v_r \neq v_{\max} \end{cases}$$

其中，$v_{\max}$ 是图中距离顶点 v_r 跳数最远的图顶点。

证明：如果不采用 ASMP 方法，当顶点 v_r 在第 *k* 步的值大于第 *k* 步收到的消息值，

即 $\delta[v_r] > msg_t^k[v_r]$，则顶点 v_r 的值将被更新为 $msg_t^k[v_r]$ 并向所有出度边顶点 $E^{out}[v_r]$ 发送消息。采用 ASMP 方法后，如果顶点 v_r 在计算处理前，其第 k 步的消息值大于第 k+1 步的消息，即 $msg_t^k[v_r] > msg_{t+1}^{k_s}[v_r]$，则第 k 步的消息将被剪枝，被剪枝的冗余消息的规模为 $|E^{out}[v_r]|$。递归的，在第 k+1 步，对于顶点 v_r 的出度边顶点 v_m，如果 $m_{t+1}^k[v_m] > m_{t+2}^{k_s}[v_m]$（ASMP 剪枝）或者 $m_{t+1}^k[v_m] \geq \delta[v_m]$（BSP 同步剪枝），那么顶点 v_m 的值，也不会被更新，被剪枝的冗余消息的规模为 $|E^{out}[v_m]|$。当距离 v_r 的最远跳数的顶点 $v_{\max}$ 被处理后，顶点 v_r 的剪枝收益计算完毕。

虽然连通域计算的图状态转换过程只有稳态和收敛态，但是其计算逻辑与单源最短路径相似。所以 ASMP 算法同样适用于连通域计算。由定理 5.1 可知，ASMP 算法的剪枝效果，依赖于 $M_{t+1}^{k_s}$ 中的消息规模，只有消息量较大时，发送给下一步的消息的缓存规模较大，才能较好的剪枝当前步的消息。因此，在稳态时，ASMP 的剪枝效果十分明显，而在扩张态和收敛态，由于消息规模较小，尤其是收缩态，下一步的消息规模通常比本步的消息少，因此，剪枝效果并不是十分明显。

我们使用跨步顶点更新（ASVU）方法模拟异步迭代执行。ASVU 方法中，消息节点 H^k 的最终消息输入集合为 $M_t^k \cup M_{t+1}^{k_s}$，其中 $M_{t+1}^{k_s}$ 将从 M_{t+1}^k 中移除，以免消息重复计算。显然，如果 $\delta[u] > msg_t^k[u] > msg_{t+1}^{k_s}[u]$，则 ASVU 可以加快最短路径值的传播速度，使迭代处理提前收敛，而且相比于 BSP 同步迭代，不会产生额外的磁盘存取开销和冗余消息。然而，由于 $M_{t+1}^{k_s}$ 仅是 M_{t+1}^k 的子集，所以 $M_{t+1}^{k_s}$ 中的消息值，不能保证是第 t+1 步的最小值。如果 $msg_{t+1}^{k_s}[u] > msg_{t+1}^k[u]$，则新生成的消息，依然是冗余消息。特别的，如果 $msg_t^k[u] > \delta[u] > msg_{t+1}^{k_s}[u]$ 或者 $u \in V_{t+1}^{k_s} \wedge u \notin V_t^k$，则对比于 BSP 同步实现方式，会产生额外的冗余消息和磁盘存取开销。

（2）跨步消息合并处理（ASMC）

以 PageRank 计算为例，跨步消息合并处理，即⊗运算⊕运算相同，可以加快消息的传播。其具体执行流程与算法 5.1 类似，只是第 6-7 行的剪枝处理换为合并处理，即如果顶点 u 的消息在 M_t^k 和 $M_{t+1}^{k_s}$ 中同时存在且，则 $msg_t^k[u] += msg_{t+1}^{k_s}[u]$，即第 t+1 步的消息 $msg_{t+1}^{k_s}[u]$ 在第 t 步被提前处理并从 $M_{t+1}^{k_s}$ 中移除，避免在第 t+1 步被重复处理。

对于增量 PageRank 计算，如果顶点 u 没有收到消息，或者收到的消息的最终合并结果为 0，则表明顶点 u 的 PageRank 值在本步不需要发生变化。因此，如果顶点 u 在第 t 步收到了消息且消息的最终合并结果不为 0，则表明顶点 u 在第 t 步将更新 PageRank 值并发送新消息。在这种情况下，将第 t+1 步的消息提前处理，不会产生额外的磁盘读取开销和冗余消息，反之，由于 u 的更新引入了最新的消息值，因而加快了 PageRank

值的传播速度，能够加快收敛。特别的，如果引入第 t+1 步的消息并与本步消息合并后，最终的结果为 0，则可以实现类似于 ASMP 的消息剪枝效果。

5.2 基于连续划分的图顶点切分方法（VCCP）

5.2.1 基于连续划分的 VCCP 方法

消息通信，是影响分布式计算框架效率的重要因素。在大图迭代处理过程中，一个大图会被分割成若$|P|$个子图并由$|P|$个任务并行处理。计算过程中，通过消息的方式，完成各个子图之间的中间结果的交换。图算法的高频迭代特点，扩大了消息通信的累积影响。5.1 小节通过基于 EBSP 模型的 Hybrid 迭代机制来约减消息规模，本小节将从数据组织结构角度设计优化措施，减少消息的网络传输规模。

PowerGraph 的 Vertex-Cut 机制，通过切分、备份顶点，能够显著降低网络通信开销，但是其实际效果，即顶点的平均备份规模，受图分割算法的制约[10]：基于 RHP 划分的随机切分（VCRHP），简单易行，划分速度快，但是因局部性差，导致效果较差。高级的启发式划分算法 Oblivious 和 Coordinated，效果较好，但是划分效率低。

本文在第三章分析了 DFS 生成图和 BFS 生成图的原始局部性，并设计了连续划分策略，以较低的划分开销，保留了原始输入图的局部性。由于 BFS 技术生成的图具有质量高、易并行特点[29,31]，并且具有水平局部性，更适合采用 Vertex-Cut 思想进行优化，因此，本节提出了基于连续划分的图顶点切分策略（VCCP）。在具体介绍 VCCP 方法之前，我们首先做如下假设：

（1）原始输入图是采用 BFS 技术爬取生成且以邻接表组织管理；

（2）图顶点标识符为数字连续编号且全局唯一；

（3）对任意顶点 v，其出度边顶点为 $adj[v]$，则 $\forall u \in adj[v]$，以顶点 u 为源顶点的邻接表记为 $adjlist[u]$，假设 v 的所有出度边顶点构成的邻接表记录集合的存储规模，不会超过任何一个分区 P_i 的图数据规模，即其可以在同一个分区内组织管理，即：

$$\sum_{\forall u \in adj[v]} byte[adjlist[u]] \le byte[P_i]$$

VCCP 的处理过程较为简单。在完成 CP 连续划分之后，各任务获得全局路由表 G，然后进入 Shuffle 阶段，完成出度边的重新分配和顶点切分操作。具体的，对于分区 P_i，依次遍历所有顶点的出度边，根据出度边目的顶点编号和全局路由表 G，判定该边所属的目的分区 $P_i^{'}$。对于分区 $P_i^{'}$，源顶点相同的边，仍然按照邻接表组织。其中，$P_i^{'}$ 上的源顶点，是分区 P_i 上对应的源顶点的一个备份。在 $P_i^{'}$ 中，以源分区为单位，组织不同源分区的图数据。而源分区 P_i 上的邻接表的出度边列表，则替换为出度边目的顶点所在的分

区编号的列表。Shuffle 阶段结束后，即进入正常的迭代计算过程。如图 5.2 所示，分区 P_1 上邻接表<1, <2, 3, 4>>，经过 Shuffle 阶段后，P_1 的邻接表变为<1, <par1, par2>>，即 1 号顶点的出度边的目的顶点分布在 1 号分区和 2 号分区。对应的，在 1 号分区和 2 号分区中，分别维护该邻接表的部分出度边：<1, <2, 3>>和<1, <4>>。则对于 1 号图顶点，共有两个备份。

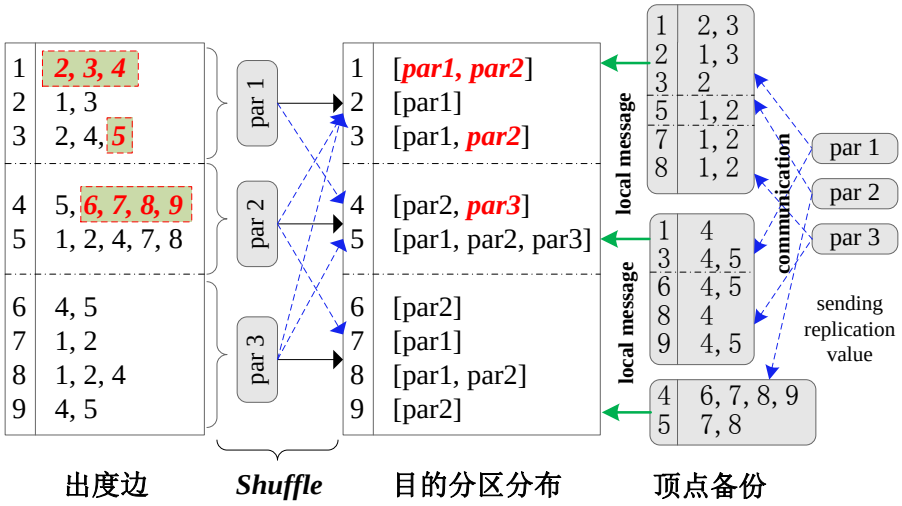


图 5.2 VCCP 方法流程示意图

Fig. 5.2 Illustration of the VCCP method

由于 VCCP 方法将原始邻接表的出度边数据分割存储到其目的图顶点所在的分区，所以迭代计算的处理流程也需要重新定义。

图顶点数据的表示方式为：<ID, Value, S>，其中，S 为扩展部分，用于记录出度边的相关统计信息。对于某些图迭代算法，需要根据出度边的完整的统计信息完成本地计算与消息发送。比如 PageRank 计算，顶点 u 发送给其目的图顶点的投票值，需要根据 u 的 PageRank 值与所有出度边总数相除，平均分配。而 VCCP 方法完成 Shuffle 阶段后，出度边列表变为“目的图顶点所在的分区列表”，其数目与出度边数目不一定相等。因此，在 Shuffle 之前，本地图顶点数据必须记录出度边的相关统计信息，S 的具体内容，可以由用户自定义。对于 PageRank 计算，S 的内容即为出度边的总数目。而对于单源最短路径计算，S 值为空。

在 VCCP 的迭代过程中，有两类消息，第一类是图顶点的备份值，需要通过网络发送给位于不同目的分区的备份顶点，另一类为正常消息，发送给出度边的目的图顶点，不需要通过网络远程传输。我们称第一类消息为“网络消息”，第二类消息为“本地消息”。在第 i 步迭代，图顶点 v 首先接收“本地消息”，然后根据顶点值 Value 和出度边统计信息 S，完成本地计算并按照邻接表中的“目的分区列表”向对应的分区，发送“网络消息”。而目的分区收到“网络消息”后，通过查找本地备份的图数据，提取出备份顶点的部分邻接表，然后根据出度边信息，生成“本地消息”并发送给本地的图顶点数

据。VCCP 技术将“本地消息”的计算过程转移到目的分区，从而极大的减少了网络通信开销。

由于备份的图数据也驻留磁盘，因此“网络消息”与目的分区的备份图数据之间，也需要进行连接操作。因为图顶点是顺序编号的且目的分区按照源分区分别组织备份的图数据，所以目的分区的备份数据，是按照图顶点编号有序存放在磁盘上。而迭代计算过程中，源分区将按照同样的顺序遍历计算并生成“网络消息”，因此，“网络消息”也是按照备份图数据的图顶点有序到达的。这种“有序性”简化了二者之间的连接操作，使目的分区的备份图数据在一次迭代过程中，仅需要完成一次磁盘存取操作即可。

如图 5.2 所示，对于 2 号分区上的第 5 号图顶点，根据“本地消息”完成其 Value 的更新后，根据用户自定义处理逻辑产生“网络消息”。对于 PageRank 计算，即将其 PageRank 值的变化与 S 中存储的总出度边数目相除，得到需要发送的“网络消息”值。对于单源最短路径计算，则直接将 Value 值作为“网络消息”值。网络消息按照目的分区发送至 1 号、2 号和 3 号分区（par1, par2, par3）。P_2 收到 P_1 发送的“网络消息”后，提取本地的备份数据<5, <1, 2>>，然后产生“本地消息”。对于 PageRank 计算，直接将收到的“网络消息”按照出度边的目的图顶点转发。对于单源最短路径计算，需要根据“网络消息”和出度边的权重生成“本地消息”，松弛出度边的目的图顶点。

5.2.2 顶点备份比例分析

对于随机 Vertex-Cut，Joseph E. Gonzalez 等人推导出一个顶点 v 的备份数目的数学期望，该值由公式 5.2 给出[10]：

$$\Im[|H(v)|]=|P|(1-(1-\frac{1}{|P|})^{|E^{out}[v]|}) \tag{5.2}$$

其中，$|P|$是该作业的任务数目（也即分区数目），而 $E^{out}[v]$ 则是顶点 v 的出度边集合。则对于输入图 G，全部顶点的备份的数学期望由公式 5.3 计算得到[10]：

$$\Im[\frac{1}{|V|}\sum_{v\in V}|H(v)|]=\frac{|P|}{|V|}\sum_{v\in V}(1-(1-\frac{1}{|P|})^{|E^{out}[v]|}) \tag{5.3}$$

定义 5.4 至定义 5.7 介绍了 VCCP 定点备份数目分析过程中的相关概念。

定义 5.4 前驱分区集合（P_{pre}）。在 *Shuffle* 操作执行之前，顶点 *v* 所在的目的分区可以通过查找全局路由表 *G* 得到，即 P_i=*getPartition(G, ID[v])*。则在 *CP* 划分策略下，分区号小于等于 *i* 的分区集合为顶点 *v* 的前驱分区集合：$P_{pre}=\{P_0, P_1, \ldots, P_i\}$。

定义 5.5 最大目的顶点编号（MDVID）。对于顶点 *v*，如果 $\exists u\in adj[v]$，$\forall x\in adj[y]$，使得 $ID[u]\geq ID[x]$，那么称 $ID[u]$为顶点 *v* 的最大目的顶点编号（MDVID）。其中 *y* 满足如下条件：P_j=*getPartition(G, ID[y])*且 $P_j\in P_{pre}$。

定义 5.6 随机分布边集（$E_{rde}^{out}[v]$）。在 *Shuffle* 操作执行之前，顶点 v 的出度边集合为 $E^{out}[v]$，那么目的顶点编号小于等于 $MDVID[v]$的出度边的集合，称为顶点 v 的随机分布边集，即 $E_{rde}^{out}[v]=\{e\,|\,DstID[e]\le MDVID[v]\wedge e\in E^{out}[v]\}$。

随机分布边集对应于 3.1.1 小节的前驱边集。由于前驱边集中的目的图顶点已经被爬取，因此对于顶点 v 而言，其分布无局部性，故本文假设随机分布边集中的目的图顶点在前驱分区集合中各个分区，是随机分布的。

定义 5.7 聚集分布边集（$E_{cde}^{out}[v]$）。在 *Shuffle* 操作执行之前，顶点 v 的出度边集合为 $E^{out}[v]$，那么目的顶点编号大于 $MDVID[v]$的出度边的集合，称为顶点 v 的随机分布边集，即 $E_{cde}^{out}[v]=\{e\,|\,DstID[e]>MDVID[v]\wedge e\in E^{out}[v]\}$。

聚集分布边集对应于 3.1.1 小节的后继边集。BFS 生成图具有水平局部性，且隶属于同一个源顶点的出度边，其分布具有密集性（详见 3.1.1 小节）。如图 5.2 所示，虚线框中的出度边，均为聚集分布边集。

定理 5.2 顶点 v 的聚集分布边集的目的图顶点 $V_{cde}[v]$，至多分布在两个分区中。

证明：定义有序顶点队列 $Q_{BFS}[v]=\{x\,|\,x\in adj[u]\wedge ID[u]<ID[v]\}$，显然 Q_{BFS} 中的顶点是 BFS 生成图的过程中，已经爬取或者待爬取的图顶点，QBFS 中顶点的入队顺序，由各顶点的出度边目的顶点的出现顺序决定。则根据定义 5.4，$\forall u\in V_{cde}[v]$，$u\notin Q_{BFS}[v]$。由于 BFS 生成图具有水平局部性，则 $V_{cde}[v]$具有原子性，即入队和出队过程中，不会被分割。根据 VCCP 方法的假设（3），$V_{cde}[v]$所属的分区的数目，为 1 或者 2.

对于图顶点 v，VCCP 方法的备份数目的数学期望由公式 5.4 计算得到：

$$\Im[|R(v)|]=|P_{pre}|(1-(1-\frac{1}{|P_{pre}|})^{|E_{rde}^{out}[v]|})+\chi \tag{5.4}$$

其中，χ 是一个附加变量，假设顶点 v 的 $V_{cde}[v]$没有跨区分布，则根据定理 5.2，当$|E_{rde}^{out}[v]|>0$ 且$|P_{pre}|<|P|$时，$\chi=1$，否则 $\chi=0$。事实上，只有部分顶点 v 的 $V_{cde}[v]$分布在两个连续的分区，因此，对于整个图 G 的所有顶点而言，跨区分布的情况至多出现$(|P|-1)$次，所以在分析一个图顶点的顶点备份数目时，忽略跨区分布的情况。但是在计算整个图的顶点的备份数目时需要考虑，见公式 5.5：

$$\Im[\frac{1}{|V|}\sum_{v\in V}|R(v)|]=\frac{1}{|V|}((|P|-1)+\sum_{v\in V}\Im[|R(v)|]) \tag{5.5}$$

定理 5.3 VCCP 方法中，顶点 v 的备份数目少于随机 Vertex-Cut 方法，即 $\Im[|R(v)|]<\Im[|H(v)|]$。

证明：对于特定点的顶点 v，其出度边和 χ 值固定，则顶点备份数目仅是关于分区数目的函数，故假设函数 $f(|P|)=\Im[|H(v)|]$，函数 $g(|P_{pre}|)=\Im[|R(v)|]$。对函数 $f(|P|)$一

阶求导（公式 5.6）可得：

$$f(|P|)' = 1-\left(1-\frac{1}{|P|}\right)^{|E^{out}[v]|-1}\left(1+\frac{|E^{out}[v]|-1}{|P|}\right) \tag{5.6}$$

二阶求导（公式 5.7）可到：

$$f(|P|)'' = \frac{1-|E^{out}[v]|}{|P|^2}\cdot\left(1-\frac{1}{|P|}\right)^{|E^{out}[v]|-2}\cdot\left(2+\frac{|E^{out}[v]|-2}{|P|}\right) \tag{5.7}$$

（1）当$1-|E^{out}[v]|<0$，即顶点 v 的出度边数目大于等于 2 时，$f(|P|)''$恒为负值，则$f(|P|)'$为单调递减函数，故$f(|P|)' \geq f(|P|)'|_{|P|\to+\infty}$。而$\lim_{|P|\to+\infty} f(|P|)' = 0$且$f(|P|)'|_{|P|=1} = 1>0$，故$f(|P|)'$恒为正值，则$f(|P|)$为单调递增函数。

（2）当$1-|E^{out}[v]|=0$，即顶点 v 的出度边的数目为 1 时，$f(|P|)' \equiv 1>0$，所以函数$f(|P|)$单调递增。

（3）$1-|E^{out}[v]|>0$，即顶点 v 无出度边时，因为$|P|\geq 1$，故$f(|P|)'' \geq 0$，当且仅当$|P|=1$时，等号成立。所以$f(|P|)'$为单调递增函数，故$f(|P|)' \geq f(|P|)'|_{|P|=1} = 1>0$，所以$f(|P|)$为单调递增函数。

综上所述，$f(|P|)$为单调递增函数，同理$g(|P_{pre}|)$也是单调递增函数。对于顶点 v，有$|P|\geq|P_{pre}|$，而且$E^{out}_{rde}[v]\subseteq E^{out}[v]$，故$f(|P|)\geq g(|P_{pre}|)$，也即 VCCP 的顶点备份数目少于随机 Vertex-Cut 方法。

定理 5.3 从理论上证明了 VCCP 方法的优异，在实验环节，我们将通过实际数据的测试，验证其有效性。

5.3 实验结果与分析

5.3.1 实验设置

实验集群由 33 台计算机组成，1 台主控节点，32 台计算节点，每台计算机的配置为：2 Intel Core i3-2100 CPUs，8GB 内存和 500GB 硬盘，磁盘转速为 7200RMP。所有计算机安装 Linux RedHat 6.0 操作系统并由 1 台 Gbps 带宽的交换机互联。实验过程中，为避免同一台计算机上不同任务之间的内存与磁盘争用影响，每个计算节点仅运行一个任务，每个任务的 JVM 配置 1GB 内存。实验采用单源最短路径计算和 PageRank 计算作为示例程序。实验数据集如表 5.2 所示。

5.3.2 基于 EBSP 模型的 Hybrid 方法性能测试

本组实验采用 20 台计算节点，启动 20 个任务，每个任务配置 2GB 内存，运行单

源最短路径计算示例程序，测试 ASMP 方法和 ASVU 方法。对于 USA-RN，同样只截取前 300 步的统计信息。两种方法均基于第 4 章的动态可调 Hash 索引实现，所以动态可调 Hash 索引将作为基准对比方案（baseline）。

表 5.2 实验数据集说明

Table 5.2 Characteristics of data sets

数据集	顶点	出度边	平均出度
Patent	3,774,768	16,518,948	4.38
S-LJ	4,847,571	68,993,773	14.23
USA-RN	23,947,347	5,833,333	0.244
Wiki-PP	5,716,808	130,160,393	22.77
Twitter	41,700,000	1,470,000,000	35.25

迭代性能分析。从图 5.3、图 5.4 和图 5.5 可知，与 baseline 方法相比，ASMP 和 ASVU 在稳态阶段能够显著提高处理效率，而在扩张态和收缩态，效果甚微。由于不同图的拓扑结构的异构性，导致状态转换过程不同，因此总体收益也不尽相同。对于 S-LJ，图的拓扑结构紧密并且连接关系分布均匀，稳态阶段的驻留时间在总体运行时间中所占的比例较大，ASMP 可以将整体运行效率提高 23%，ASVU 方法可以提高约 8.34%。对于 USA-RN 和 Wiki-PP，前者扩张态耗时过长，后者的收缩态耗时过长，因此总体性能改善不大。

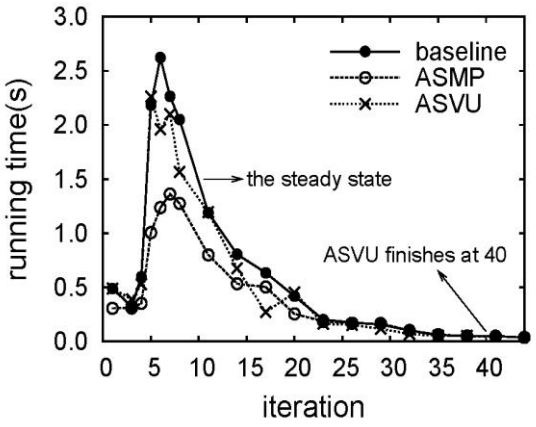


图 5.3 S-LJ 迭代性能

Fig. 5.3 The running time per iteration of S-L

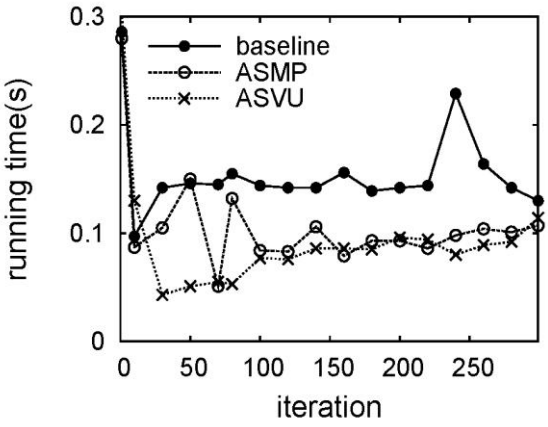


图 5.4 USA-RN 迭代性能

Fig. 5.4 The running time per iteration of USA-RN

收敛速度分析。图 5.3 中，对于 S-LJ，ASVU 方法仅需迭代 40 步即收敛，而 ASMP 方法需要 44 步。类似的，在图 5.5 中，对于 Wiki-PP 数据集，ASVU 方法迭代 288 步，而 ASMP 方法需要 290 步。这是因为 ASVU 方法可以加快消息的传播速度，从而缩短迭代步数，即从运行所需的超级步角度看，可以加速收敛，证明了异步迭代的有效性。

消息与激活的图顶点规模分析。对于三个不同的数据集，我们分别收集其在 baseline 方法下消息接收规模的最大的超级步的统计信息，分析 ASMP 和 ASVU 方法对于消息与激活的图顶点规模的影响。图 5.6 展示了接收消息规模的不同，对于 S-LJ，ASMP 的

消息剪枝比例可以达到 55.6%，对于 Wiki-PP，也可以达到 54.6%，而 USA-RN 方法，高达 68%。对于接收消息的剪枝效果，会直接影响当前超级步的激活顶点规模，如图 5.7 所示，三种数据集的激活顶点规模分别降低了 39%、68%和 40%，而激活顶点规模的降低，可以减少本地图数据的磁盘存取开销。进一步的，也会降低新生成的消息规模。如图 5.8 所示，本步发送的消息规模分别降低了 46%、68%和 45%，从而节省了网络通信开销。发送消息规模的具体剪枝效果，受图的平均出度的影响，由于 S-LJ 和 Wiki-PP 的出度较高，因此剪枝比例高于激活顶点规模的剪枝比例，而 USA-RN 的出度较低，故剪枝比例与激活顶点规模的比例相同。

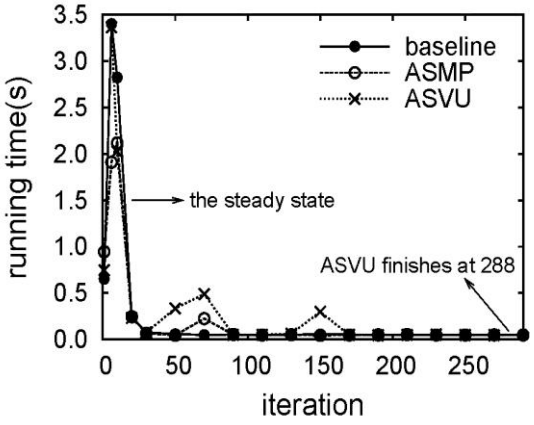


图 5.5 Wiki-PP 迭代性能

Fig. 5.5 The running time per iteration of Wiki-PP

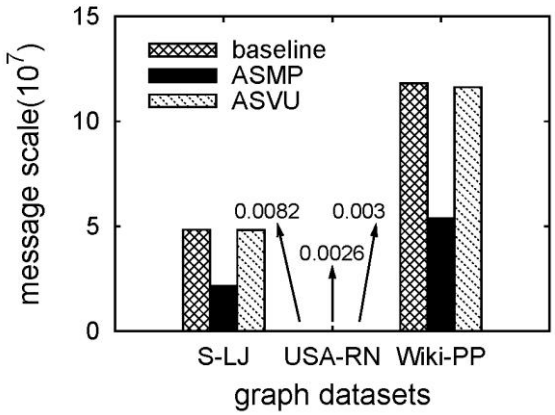


图 5.6 接收消息规模

Fig. 5.6 The scale of receiving messages

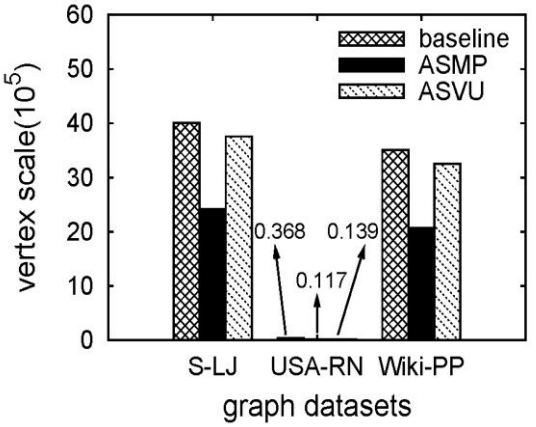


图 5.7 激活的图顶点规模

Fig. 5.7 The scale of active vertices

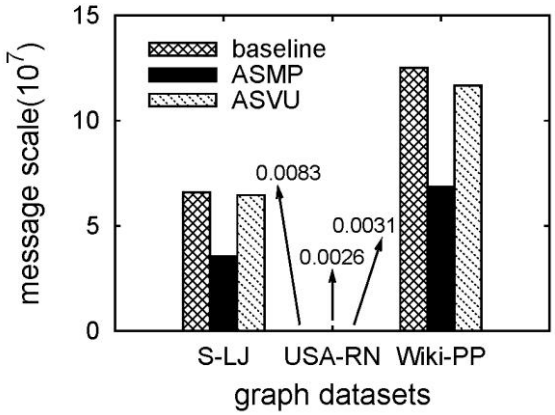


图 5.8 发送消息规模

Fig. 5.8 The scale of sending messages

从迭代性能分析结果看，虽然三种数据集的剪枝比例均比较高，尤其是 USA-RN，但只有 S-LJ 的总体性能得到了较大的改善。对于 USA-RN 图，其平均出度低，一直处于扩张态，消息基数和激活顶点规模小，导致 Warm-Up 固定开销大，所以优化效果不明显。对于 Wiki-PP 图，一方面其拓扑结构存在严重的幂律偏斜（某些顶点的重要性远高于其它顶点，它们收敛后，大部分顶点即会收敛），导致扩张态和稳态时间较短，另一方面图的直径大，故收敛态持续时间较长，而收敛态的消息基数和激活顶点规模小，

ASMP 方法优化效果有限，因此总体性能改善也不明显。

ASVU 方法将下一个超级步的消息直接作为本步消息处理，虽然也有 ASMP 的剪枝效果，但同时引入了大量新消息（详见 5.1.4 小节），故消息接收规模与 baseline 方法相近，进一步的，激活顶点规模和消息发送规模与 baseline 相近，所以虽然 ASVU 加快了消息传播速度，但是对于磁盘存取和网络通信的剪枝效果不佳，因此总体性能的优化效果不如 ASMP。

5.3.3 基于连续划分的 VCCP 方法性能测试

本组实验采用 32 台计算节点，最多启动 32 个任务，运行 PageRank，迭代 10 步。

与 PowerGraph 的对比分析。VCCP 是基于 CP 划分的 Vertex-Cut 改进方法，利用图的原始局部性，在保证顶点备份规模较小的同时减少数据预处理时间。实验数据集采用 Twitter，出度边具有幂律分布特点[38]。实验对比的方法包括：VCRHP，PowerGraph 提出的并行启发式划分算法（Oblivious 与 Coordinated）和本文提出的 VCCP 方法。图 5.9 展示了四种方法随任务数目的变化，顶点备份数目的变化趋势，图 5.10 展示了数据预处理时间。数据预处理时间包括数据加载、数据划分和 Vertex-Cut 三部分。显然，对于 VCRHP、Oblivious 和 Coornidated 而言，随着算法复杂度的提高，顶点备份数目呈减少趋势，但是数据预处理时间呈上升趋势。而 VCCP 方法，其顶点备份数目接近 Coordinated 方法，但是预处理时间与 VCRHP 方法接近。

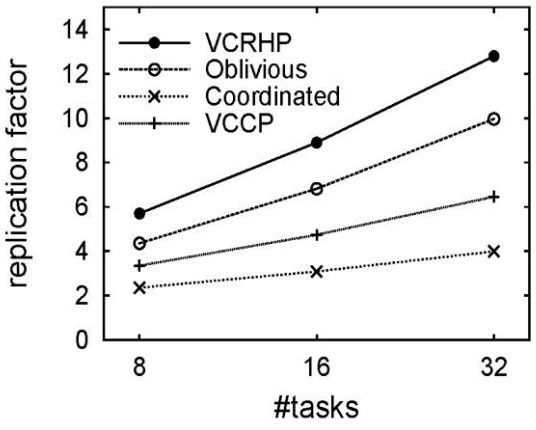


图 5.9 顶点备份数目

Fig. 5.9 The number of backup vertices

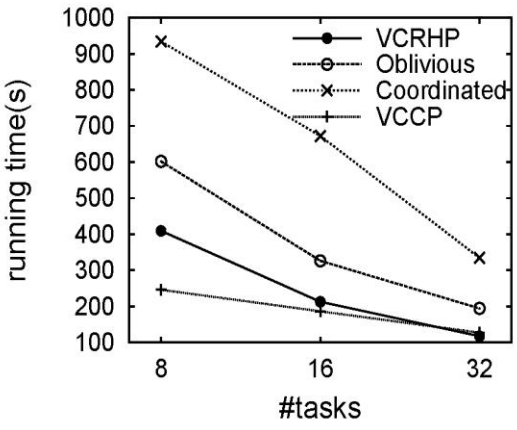


图 5.10 数据预处理时间

Fig. 5.10 The running time of preprocessing

VCCP 性能分析。图 5.11、图 5.12 和图 5.13 分别从数据预处理时间、网络通信规模和总迭代处理时间三个方面分析了 RHP 方法、CP 方法和 VCCP 方法的性能。本组实验测试了 Patent、S-LJ 和 Wiki-PP 三个数据集，启动 10 个任务。从图 5.11 可知，由于 VCCP 方法存在 Shuffle 过程，因此其预处理时间略高于 CP 方法。VCCP 的 Shuffle 过程中，仅有跨分区出度边需要进行网络通信，由于 CP 方法保留了图的原始局部性，RHP

方法无局部性，所以 VCCP 的 Shuffle 通信量低于 RHP 方法，故其预处理时间低于 RHP 方法。特别的，对于 Wiki-PP，其加速比为 1.8（CP 方法为 2.12）。图 5.12 为网络通信规模测试，与 RHP 方法相比，VCCP 在三种数据集上的剪枝比例为 71%、80%和 87%，而 CP 方法的剪枝比例分别只有 7%、25%和 20%。VCCP 通过顶点备份，可以显著降低网络通信规模，其剪枝效果与图的平均出度基本成正比。图 5.13 展示了总体迭代时间。对于 S-LJ 和 Wiki-PP，相比于 CP 方法，VCCP 可以提高效率 34%和 54%，但是对于 Patent，由于平均出度较低，消息剪枝效果有限，无法抵消 VCCP 引入的 Shuffle 额外开销，因此整体运行时间略有升高。

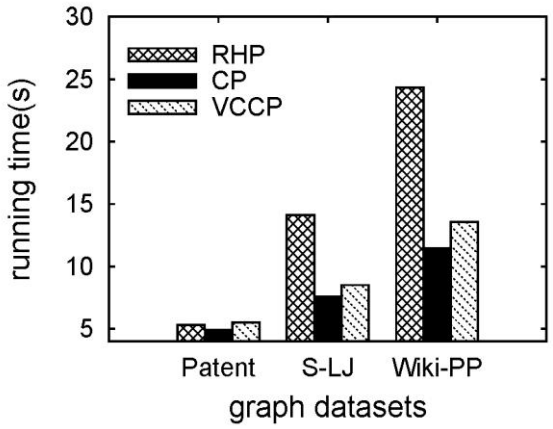


图 5.11 数据预处理时间

Fig. 5.11 The running time of preprocessing

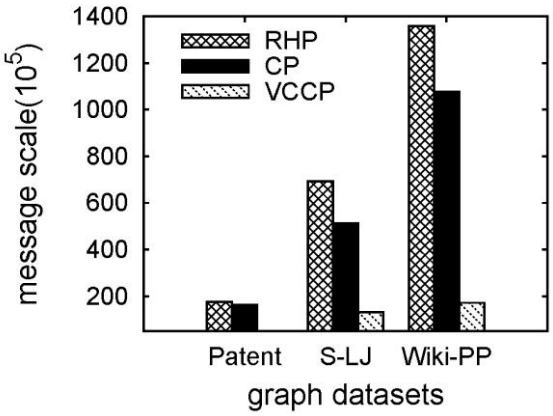


图 5.12 网络通信规模

Fig. 5.12 The scale of network messages

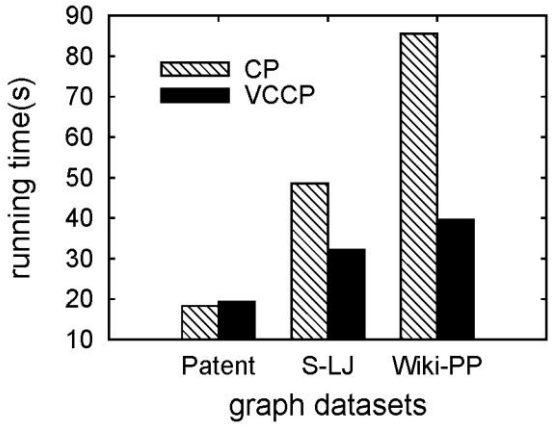


图 5.13 总迭代处理时间

Fig. 5.13 The overall running time

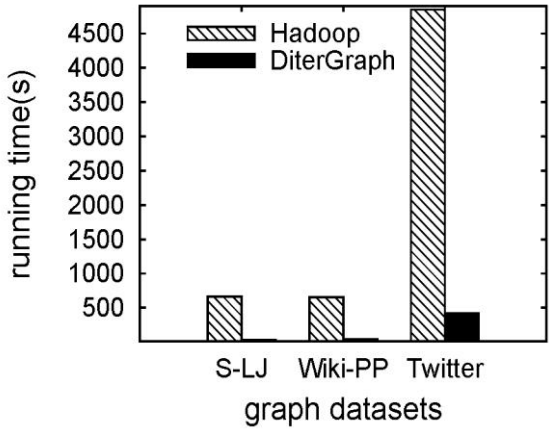


图 5.14 DiterGraph vs. Hadoop

Fig. 5.14 DiterGraph vs. Hadoop

DiterGraph 与 Hadoop 的对比分析。我们在 DiterGraph 原型系统上实现了 VCCP 方法，并与 Hadoop 进行了对比实验。如图 5.14 所示，在 S-LJ、Wiki-PP 和 Twitter 数据集上，DiterGraph 的运行效率分别为 Hadoop 的 21.85 倍、16 倍和 11.57 倍。DiterGraph 的处理性能明显优于 Hadoop，但可扩展性随数据规模的增加而略有降低。

5.4 本章小结

本章针对分布式图处理的消息通信进行优化处理，提出了基于 EBSP 模型的 Hybrid 方法，能够支持消息跨步处理和跨步合并。此外，在 CP 连续划分基础上，提出了 VCCP 方法减少消息的网络传送规模。

第6章 DiterGraph 原型系统

为高效处理大规模图的增量迭代计算并验证数据划分策略、磁盘索引策略和消息优化策略的有效性，本文设计了原型系统 DiterGraph(Disk Iteration Graph)。本章将介绍 DiterGraph 系统的主要架构、功能模块以及部署和使用方法。

6.1 系统简介

DiterGraph 是基于 Master-Slave 架构的分布式大图迭代处理系统，分为应用层、计算层和存储层。应用层为支持的增量迭代图处理应用，如最短路径计算、PageRank 计算、随机游走和连通域计算等。与 Hadoop 类似，DiterGraph 主体框架由计算层和存储层构成。DiterGraph 的磁盘存储系统由全局的分布式存储系统和各计算节点的本地存储系统组成。目前，分布式存储系统采用 HDFS。计算层由一个主控节点管理所有的计算节点。主控节点用于客户端交互、作业管理、集群管理和容错管理等。而计算节点则维护具体的任务，提供本地磁盘存储和索引以及消息收发等功能。用户提交的图处理作业，将被切分为若干任务在各计算节点上并行处理。同一个作业的所有任务由作业控制中心管理，而作业控制中心均位于主控节点。DiterGraph 的主要特点如下：

（1）采用 CP 划分策略，保留原始输入图的局部性，减少数据划分的网络通信开销；

（2）提供高效的图顶点连续编码功能，如果原始输入图的顶点为字符串，需要调用此功能对图顶点进行连续编码，以减少磁盘存取和网络通信开销，便于设计高效索引；

（3）支持磁盘存储并设计了高效的哈希索引（静态索引和动态索引），在提高系统数据能力的同时，保证计算效率；

（4）在 EBSP 基础上，支持 Hybrid 迭代处理机制，提供 ASMP 和 ASMC 功能；

（5）在 CP 划分基础上，利用图的原始局部性，融合 Vertex-Cut 技术，提供 VCCP 处理机制，优化消息通信规模。

DiterGraph 系统的主要功能模块如图 6.1 所示，主要包括 CLI/API，核心计算组件（计算预处理、迭代计算等），分布式存储系统和管理工具，外围服务模块的详细功能介绍如下：

（1）API：DiterGraph 提供的用户编程接口，用于实现特定的作业处理逻辑；

（2）CLI：用户与 DiterGraph 系统的交互接口，用于提交、管理作业，管理 DiterGraph 系统并提供相关信息的查询功能；

（3）分布式存储系统：提供分布式存储功能，存放原始图数据和处理结果，目前仅支持 HDFS，通过扩展输入输出接口，可以支持 HBase 和其它数据库系统；

（4）配置部署：用于在集群上快速自动部署 DiterGraph 系统以及分布式存储系统，如 HDFS，同时，提供 DiterGraph 和 HDFS 的相关参数的快捷设置功能；

（5）进度监控：提供可视化的作业进度监控工具，监控整个作业的运行进度和该作业中各个任务的运行进度并提供自动预警功能，对于进度缓慢的任务或发生故障的任务能够动态提示，提供良好的用户交互界面，此外还提供统计已运行时间和预估结束时间等功能；

（6）性能监控：目前仅提供各任务的内存使用情况的动态统计汇报功能；

（7）日志管理：采用 Log4j 日志管理系统管理 DiterGraph 集群和作业的运行日志信息；

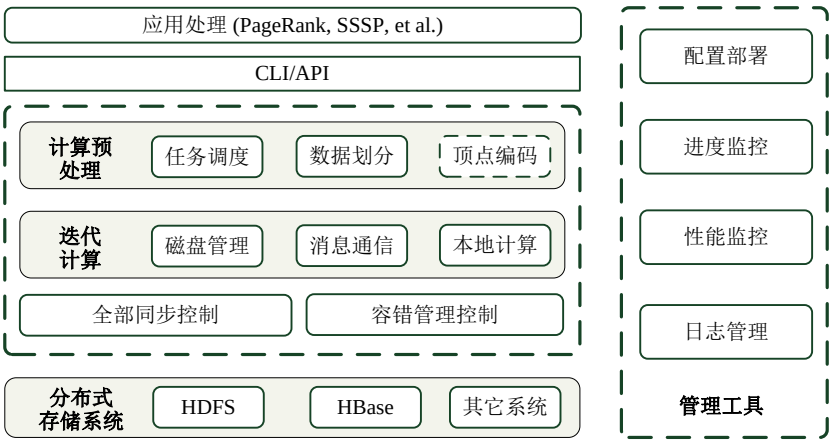


图 6.1 DiterGraph 系统功能组件

Fig. 6.1 The components of DiterGraph

DiterGraph 系统的核心组件包括计算预处理、迭代计算、全局同步控制和容错管理控制，其包含的具体功能模块如下：

（1）任务调度：用户提交的作业将在主控节点建立作业控制中心并初始化该作业的任务，然后放入作业等待队列，任务调度器根据作业优先级和 FCFS 原则调度作业，对于选定作业的任务，则根据数据本地化和负载均衡原则选派计算节点；

（2）数据划分：一个作业的所有任务调度执行后，首先从 HDFS 加载图数据并完成数据划分操作，DiterGraph 目前提供 CP 划分策略；

（3）顶点编码：可选操作，如果输入图的顶点已经连续编码，则跳过此模块，否则需要调用该模块，将输入图的顶点进行连续编码；

（4）磁盘管理：即各任务在计算节点的本地磁盘管理，包括列存储模式和哈希索引机制；

（5）消息通信：完成各任务的消息收发和组织管理操作，DiterGraph 提供 Hybrid 迭代优化和 VCCP 优化技术；

（6）本地计算：完成迭代过程中，接收消息的处理、图顶点值的更新并产生新的

消息；

（7）全局同步控制：提供分布式任务的同步管理，用于协调各任务的进度；

（8）容错控制管理：提供图数据的 checkpoint 功能，当迭代过程中发生故障时，可以从最近的检查点恢复迭代。

6.2 系统部署及使用方法

6.2.1 系统部署

DiterGraph 系统使用 Java 实现，需要依赖于第三方的分布式存储系统 HDFS，因此，部署 DiterGraph 之前，需要安装 1.6.0 以上版本的 JDK 和 Hadoop 系统。此外，底层操作系统必须为 Linux。

关于 JDK 和 Hadoop 的配置与安装方法，这里不再赘述。DiterGraph 系统的配置及安装方法与 Hadoop 类似。为保证 DiterGraph 的正常运行，需要配置 ditergraph-env.sh、workers 和 ditergraph-site.xml 三个文件。ditergraph-env.sh 文件用于指定系统启动时的命令参数，用户至少需要指定系统所在计算节点的 JDK 安装路径，例如：export JAVA_HOME=/usr/java/jdk1.6.0_23。workers 文件用于指定 DiterGraph 系统中的计算节点，每一行指定一个计算节点的网络域名。ditergraph-site.xml 文件是系统配置的核心文件，主要配置内容及说明见表 6.1。更加详细的配置选择，可以参考 ditergraph-default.xml 文件。

表 6.1 ditergraph-site.xml 文件的详细配置

Table 6.1 The detail configuration of ditergraph-site.xml

配置属性	属性值示例	备注
ditergraph.master.address	hadoop02:40000	主控节点服务端口
fs.default.name	hdfs://hadoop02:9000	指定 HDFS 服务端口
ditergraph.child.java.mem	-Xmx2048m	JVM 内存大小设置
ditergraph.task.max	2	最大任务槽/计算节点
ditergraph.local.dir	/tmp/data	本地数据存储目录

用户完成配置操作后，只需将 DiterGraph 的系统安装包（即 ditergraph-0.1 文件夹下的所有内容）拷贝到集群上的所有物理机器上即可。DiterGraph 的计算节点由 workers 文件指定，而主控节点，由配置文件中的“ditergraph.master.address”属性指定，本例中，为 LENOVO 域名所对应的物理机器。安装包复制完毕后，为启动管理方便，用户可以配置/ect/profile 文件，在其中添加如下内容：

export DITERGRAPH_HOME=/usr/lily/ditergraph-0.1

export DITERGRAPH _CONF_DIR=/usr/lily/ ditergraph -0.1/conf

export PATH=$PATH:$ DITERGRAPH _HOME/bin

所有准备工作完成后，首先启动 HDFS 并确保 HDFS 已经推出安全模式，然后在主控节点 LENOVO 的命令行执行“start-ditergraph.sh”即可启动 DiterGraph 计算系统。系统附带 PageRank 计算和单源最短路径的示例程序，用户可以在 HDFS 上准备输入数据，然后执行“ditergraph jar ditergraph-examples.jar pagerank args”来运行示例程序。

6.2.2 用户编程指导

DiterGraph 的计算处理是以图顶点为中心完成的。用户可以通过 API 接口自定义处理逻辑。具体的，用户需要继承两个类：UserTool.java 和 BSP.java，重载其中的部分方法。UserTool 类是辅助处理类，主要包括图数据和消息数据的自定义封装类。而 BSP.java 用于自定义逻辑处理流程，用户必须重载 compute 方法，该方法在每次迭代时，针对每个图顶点至多调用一次，用于接收消息、完成本地计算和产生新的消息。具体的使用方法请参考系统附带的 PageRank 和单源最短路径计算程序的源代码。

6.2.3 可视化管理工具

DiterGraph 提供 CLI 接口管理作业与集群状态，为便于用户使用，还提供了可视化的操作界面。

图 6.2 为管理工具面板，用户可以在左上角输入 Hadoop 集群和 DiterGraph 集群的主控节点的域名以及服务端口，以便管理工具能够正常连接集群。HadoopPath 和 DiterGraphPath 用于指定安装文件在物理集群上的安装目录。WorkerList 为集群中物理节点列表，可以通过右上角的面板进行添加与删除管理。“Deploy Hadoop&DiterGraph”面板项用于自动部署 Hadoop 和 DiterGraph。用户可以在“User-Defined Command”中输入自定义的 Linux 命令，执行结果可以在左下角的“Operation Result”中查看。

DiterGraph 部署完毕后，如果需要更改其配置参数，可以通过主菜单栏的“Configuration”->“Configurate DiterGraph”完成。如图 6.3 所示，“Value”被修改后，通过“Update”即可更新集群中的配置参数。其中，各物理机器的 JDK 路径和计算节点的可达任务槽数量需要根据图 6.2 中的“WorkerList”的对应值完成配置。

对于已经提交的作业，用户可以在控制台查看简略的状态信息。如图 6.4 所示，控制台输出当前的处理状态以及该状态的进度。状态分为三类：LoadGraph（预处理）、SuperStep（正常迭代计算）和 SaveGraph（保存计算结果），各状态的进度，包括最慢任务的进度和最快任务的进度。

用户也可以通过可视化界面查看详细的状态信息，如图 6.5 所示，可以展示作业的总体运行进度和所有任务在当前迭代中的运行进度。

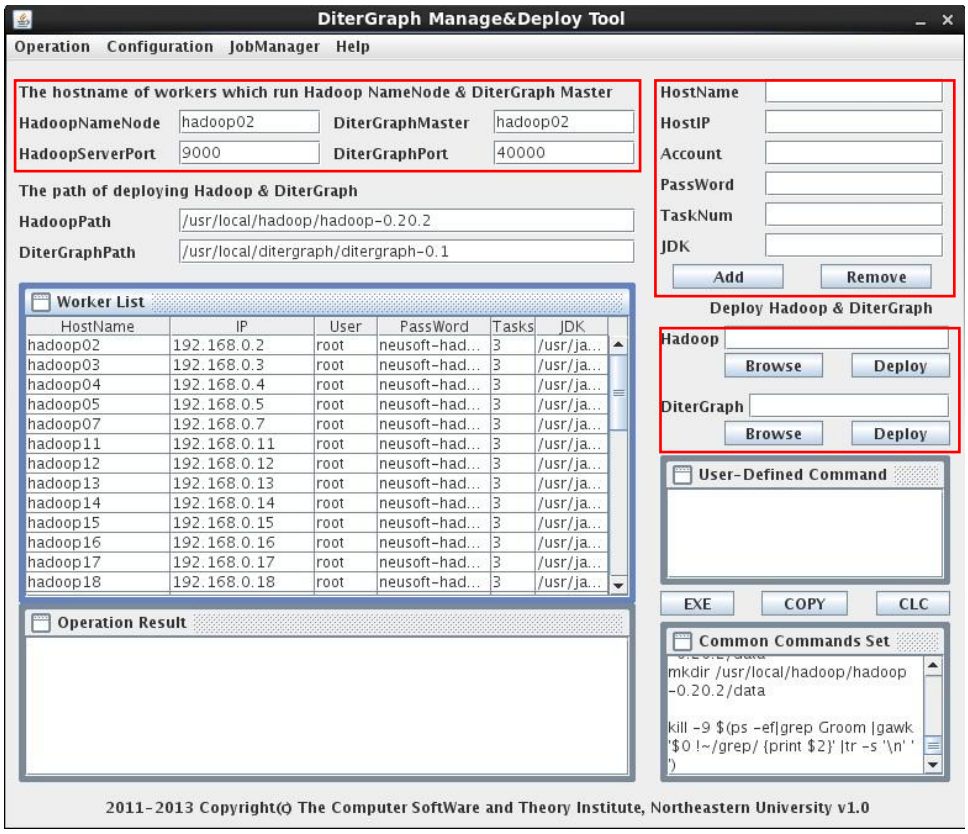


图 6.2 管理工具面板

Fig. 6.2 The panel of the management tool

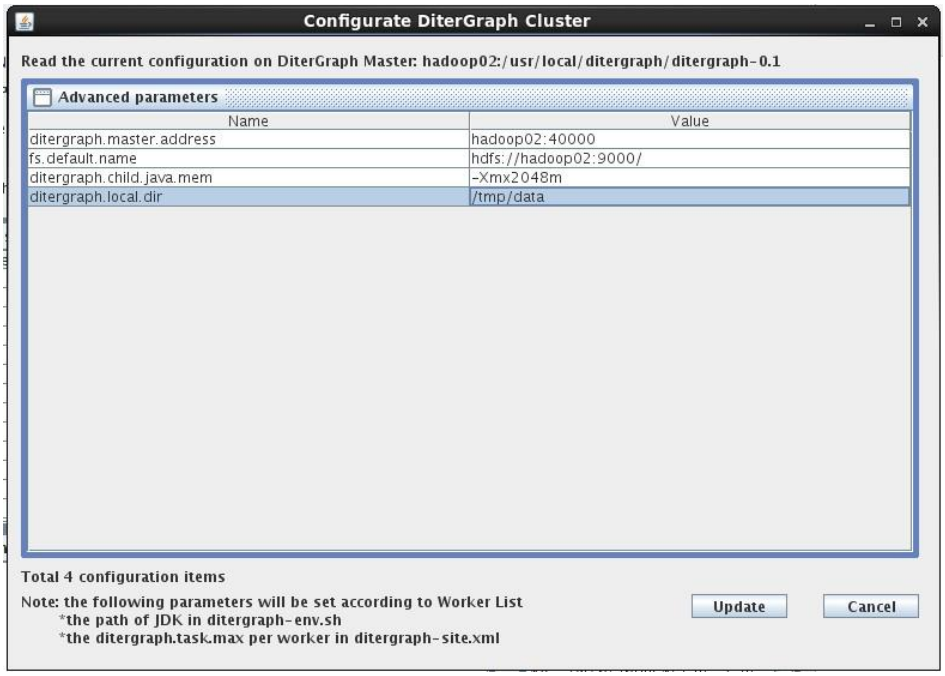


图 6.3 DiterGraph 的可视化配置选项

Fig. 6.3 The configuration panel of DiterGraph

```
12/11/06 18:40:17 : [LoadGraph] --      min:94% max:100%
12/11/06 18:40:20 : [LoadGraph] --      min:95% max:100%
12/11/06 18:40:23 : [LoadGraph] --      min:96% max:100%
12/11/06 18:40:26 : [LoadGraph] --      min:97% max:100%
12/11/06 18:40:29 : [LoadGraph] --      min:98% max:100%
12/11/06 18:40:35 : [LoadGraph] --      min:98% max:100%
12/11/06 18:40:38 : [LoadGraph] --      min:98% max:100%
12/11/06 18:40:44 : [SuperStep] 1       min:0%  max:0%
12/11/06 18:40:47 : [SuperStep] 1       min:0%  max:20%
12/11/06 18:40:50 : [SuperStep] 2       min:0%  max:20%
12/11/06 18:40:53 : [SuperStep] 3       min:0%  max:0%
```

图 6.4 作业的控制台监控结果

Fig. 6.4 The console of monitoring a job

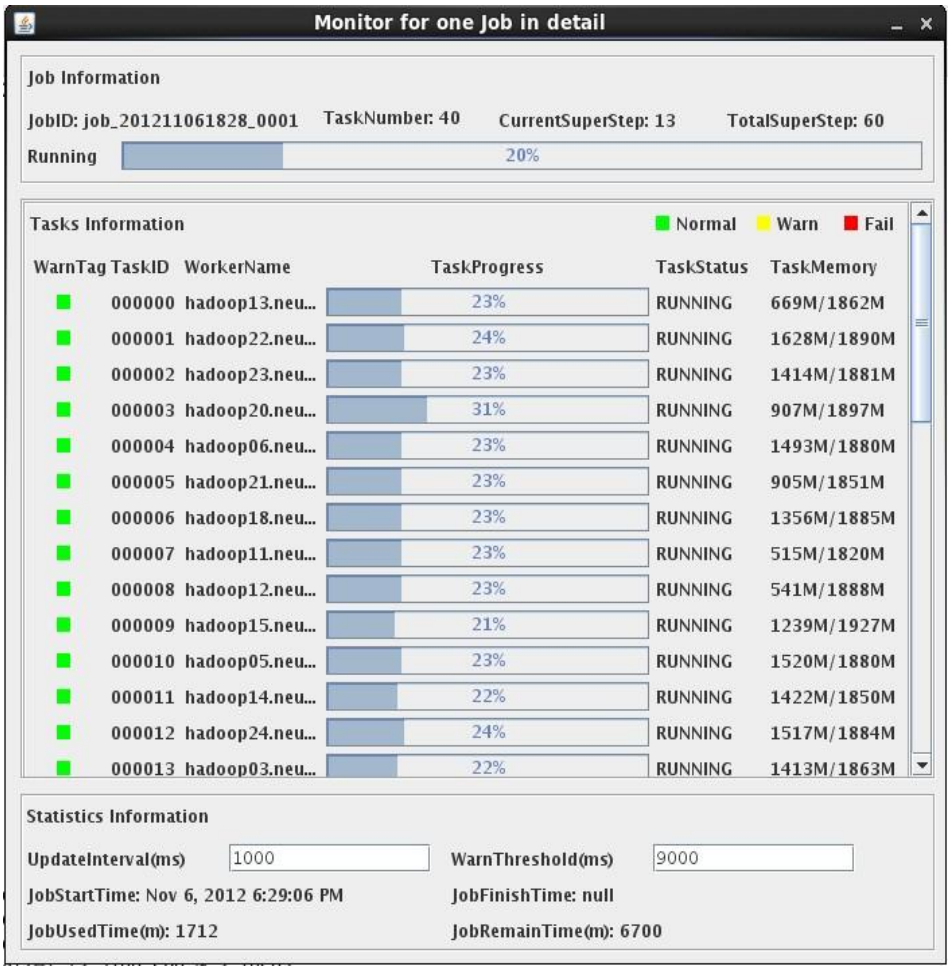


图 6.5 作业的可视化监控界面

Fig. 6.5 The panel of monitoring a job

其中，任务的运行进度利用其已经处理的图顶点数目和该任务负责的图顶点总数目的比值估算。而作业的整体运行进度利用已经完成的迭代步数和总迭代步数的比值估算。监控工具不断累计已完成的超级步的运行时间并据此估算剩余超级步的结束时间。如果某个任务的运行进度超过一定时间阈值而没有更新，则该任务的运行状态变为黄色警告，即该任务可能发生故障。如果任务在警告期间，进度被更新，则自动撤销警告，转为绿色正常状态。如果任务失败，则指示标志变为红色状态。此外，监控工具动态收集并显示各个任务的内存使用状况，使用户能够及时了解各任务的内存占用情况。

6.3 本章小结

DiterGraph 是本文针对大规模图增量迭代处理而开发的原型系统，用于实现大规模图的高效处理并验证本文提出的数据划分、磁盘索引和消息优化机制。本章首先介绍了 DiterGraph 的系统架构和主要功能模块，然后说明了其分布式部署要求和具体的使用方法，描述了 DiterGraph 的可视化管理工具的使用。

第7章 总结与展望

7.1 本文的主要贡献与结论

近年来，数据库三大会议 VLDB、SIGMOD、ICDE 中，有关大图查询处理和图数据管理方面的文章日益增多。2010 年起国际上著名的数据库学者发起了图数据管理研讨会(GDM)，并从 2011 年起与 ICDE 大会联合召开。图处理，尤其是大图的迭代计算，已经成为学术界和工业界的研究热点。本文针对大图的增量迭代计算，从数据划分、磁盘索引和消息优化三个方面进行了研究并设计实现了 DiterGraph 原型系统，用于支持大规模图的增量迭代处理。本文的主要贡献如下：

（1）采用连续划分（CP）策略，在原始图的局部性保留、数据划分开销两方面均优于随机Hash划分（RHP），同时，为提供消息路由寻址服务，设计了基于Hadoop的连续编码策略和基于DHT的Hybrid-*MT*连续编码策略。图顶点的编码，还可以节省磁盘存取和网络通信开销，便于设计高效索引；

（2）设计了基于列存储模型的图数据磁盘存储结构，提出了图增量迭代过程的状态转换模型，并以此为基础，设计了基于Markov模型的动态可调Hash索引，能够根据图迭代过程的状态转换而动态优化存盘存取的数据规模；

（3）提出了基于EBSP模型的Hybrid迭代机制，支持消息跨步剪枝（ASMP，可以约减消息规模）或跨步合并（ASMC，可以加快消息传播速度）。

（4）基于CP划分策略和Vertex-Cut技术，提出了VCCP方法，利用BFS生成图的原始局部性，以较低的预处理代价获得了较好的消息优化效果。

（6）设计实现了 DiterGraph 原型系统，DiterGraph 实现了本文的 CP 划分方法、基于 DHT 的图顶点 Hybrid-*MT* 连续编码方法、磁盘存储与动态索引、Hybrid 迭代机制和 VCCP 方法，因此能够高效支持大规模图的分布式磁盘迭代处理。

7.2 未来工作

大图的增量迭代处理已经吸引了越来越多的科研工作者和商业公司的注意，关于数据划分和消息优化的研究成果不断涌现，Google、微软等大型公司和 Apache 开源社区均投入了大量的资源研究大图处理系统。分析当前的最新研究成果和本文的相关技术与实验结果，下一步的研究工作主要包括三个方面：

（1）VCCP技术优化。VCCP能够显著降低消息通信规模，但是存在以下问题：(A)所有出度边全部被切分。实际上，对于顶点v，如果目的分区仅包含顶点v的一条出度边，则其顶点和出度边的备份是无意义的，增大了额外维护开销，所以应对出度边进行部分

切分，根据代价收益模型，分析切分的阈值条件；(B) 负载不均衡。CP划分按照计算负载进行了均衡性调整，但是，如果采用VCCP方法将生成消息的操作从发送端转移到接收端，则由于入度分布的幂律偏斜偏斜，会导致新的负载不均衡，所以应该改进VCCP方法，在Shuffle阶段，考虑负载均衡，平衡发送端和接收端的负载。

（2）迭代机制优化。完全异步迭代可以加快消息传播过程，但对于高入度顶点，会造成顶点频繁更新并产生大量冗余消息，因此可以考虑将图顶点分为两类：低入度顶点，采用异步迭代机制；高入度顶点，采用BSP同步迭代机制。

（3）根据实验结果，DiterGraph的处理效率虽然明显由于Hadoop，但是随着数据规模的增长，加速比逐渐降低，所以应该进一步优化DiterGraph系统，改善其可扩展性。

参考文献

1. Lu J. Big Data Management - Challenges and Opportunities - an Incomplete Survey[R], Beijing: Tsinghua University, 2012.
2. Che D, Safran M, Peng Z. From Big Data to Big Data Mining[A], BDMA, DASFAA WorkShops[C], 2013, unpublished.
3. 于戈, 谷峪, 鲍玉斌, et al. 云计算环境下的大规模图数据处理技术[J], 计算机学报, 2011, 34(10): 1753-1767.
4. Page L, Brin S, Motwani R, et al. The PageRank Citation Ranking: Bringing Order to the Web[R], Technical Report, Stanford University, 1999.
5. Zhang Y, Gao Q, Gao L, et al. Accelerate Large-Scale Iterative Computation through Asynchronous Accumulative Updates[A], In Proc. of ScienceCloud[C], 2012, 13-22.
6. Ewen S, Tzoumas K, Kaufmann M, et al. Spinning Fast Iterative Data Flows[J], PVLDB, 2012, 5(11): 1268-1279.
7. Malewicz G, Austern M H, Bik A J C, et al. Pregel: A System for Large-Scale Graph Processing[A], In Proc. of SIGMOD[C], 2010, 135-146.
8. Apache Incubator Giraph[EB/OL], http://incubator.apache.org/giraph/, 2013.
9. Salihoglu S, Widom J. GPS: A Graph Processing System[R], Technical Report, Stanford University, 2012.
10. Gonzalez J, Low Y, Gu H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs[A], In Proc. of OSDI[C], 2012, 17-30.
11. Apache Hadoop[EB/OL], http://hadoop.apache.org/, 2013.
12. Lang W, Patel J M. Energy Management for MapReduce Clusters[J], PVLDB, 2010, 3(1-2): 129-139.
13. 王军, 冯铃, 薛文伟. 服务器与集群系统节能技术研究[J], 软件, 2011, 32(2): 4-8.
14. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters[M], Comm. of the ACM-50th anniversary issue, 2008, 51(1): 107-113.
15. Bu Y, Howe B, Balazinska M, et al. HaLoop: Efficient Iterative Data Processing on Large Clusters[J], PVLDB, 2010, 3(1-2): 285-296.
16. Ekanayake J, Li H, Zhang B, et al. Twister: A Runtime for Iterative MapReduce[A], In Proc. of HPDC[C], 2010, 810-818.
17. Valiant L G. A Bridging Model for Parallel Computation[M], Comm. of the ACM, 1990, 33(8): 103-111.
18. Seo S, Yoon E J, Kim J, et al. HAMA: An Efficient Matrix Computation with the MapReduce Framework[A], CloudCom[C], 2010, 721-726.
19. METIS[EB/OL], http://glaros.dtc.umn.edu/gkhome/views/metis, 2013.

20. Apache Hama[EB/OL], http://hama.apache.org/, 2013.
21. Shao B, Wang H, Li Y. The Trinity Graph Engine[R], Technical Report, Microsoft Research, 2012.
22. Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster Computing with Working Sets[A], In Proc. of HotCloud[C], 2010, 10-17.
23. Stanton I, Kliot G. Streaming Graph Partitioning for Large Distributed Graphs[A], In Proc. of KDD[C], 2012, 1222-1230.
24. Barnard S T. PMRSB: Parallel Multilevel Recursive Spectral Bisection[A], In Proc. of Supercomputing[C], 1995, 27.
25. Hendrickson B, Leland R. A Multilevel Algorithm for Partitioning Graphs[R], Technical Report, Sandia National Laboratories, 1995.
26. Neo4j, the Graph Database, http://www.neo4j.org/, 2013.
27. Cho J, Garcia-Molina H, Page L. Efficient Crawling Through URL Ordering[J], Computer Networks and ISDN Systems, 1998, 30(1-7): 161-172.
28. Cho J. Crawling the Web: Discovery and Maintenance of Large-Scale Web Data[D], Standford University, 2001.
29. Najork M, Wiener J L. Breadth-First Search Crawling Yields High-Quality Pages[A], In Proc. of WWW[C], 2010, 114-118.
30. Abiteboul S, Preda M, Cobena G. Adaptive On-Line Page Importance Computation[A], In Proc. of WWW[C], 2003, 280-290.
31. Boldi P, Codenotti B, Santini M, et al. UbiCrawler: a scalable fully distributed web crawler[J], Software-Practive & Experience, 2004, 34(8): 711-726.
32. 王晓东. 算法设计与分析[M], 北京:清华大学出版社, 2003.
33. Kitsuregawa M, Nakayama M, Takagi M. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method[A], In Proc. of VLDB[C], 1989, 257-266.
34. Li B, Mazur E, Diao Y, et al. A Platform for Scalable One-Pass Analytics using MapReduce[A], In Proc. of SIGMOD[C], 2011, 985-996.
35. Stanford Large Netework Dataset Collection[EB/OL], http://snap.stanford.edu/data/, 2013.
36. Using the Wikipedia link dataset[EB/OL], http://haselgrove.id.au/wikipedia.htm, 2013.
37. Twitter dataset[EB/OL], http://an.kaist.ac.kr/~haewoon/release/twitter_social_graph/, 2013.
38. Kwak H, Lee C, Park H, et al. What is Twitter, a Social Network or a News Media?[A], In Proc. of WWW[C], 2010, 591-600.
39. Kamvar S, Haveliwala T, Golub G. Adaptive methods for the computation of PageRank[J], Linear Algebra and its Applications, 2004, 386: 51-65.
40. 9th DIMACS Implementation Challenge-Shortest Paths[EB/OL],

http://www.dis.uniroma1.it/challenge9/download.shtml, 2013.

致 谢

首先诚挚的感谢指导老师于戈教授。两年的时间里，您曾在百忙之中抽出时间，不辞辛劳为我审阅论文至深夜；您多次为我提供学术交流机会，让我开阔视野；您在生活上对我的照顾无微不至。往事的点点滴滴，汇聚在一起，令我的心中久久不能平静。辛劳多年，您已桃李遍天下，我非常荣幸能够成为其中的一员。您对我的谆谆教诲是我一生最大的财富。您严谨的学术作风和宽厚的处事风格，是我学习的典范。

感谢计算机软件与理论研究所鲍玉斌老师和谷峪老师两年来对我的倾力培养，锻炼了我的项目实战经验和学术素养，磨练了我的毅力。怀念项目申请时的忙乱与压力，怀念第一次与客户沟通时的紧张与不安，怀念项目验收时的忙碌与兴奋，怀念讨论问题时的激烈，怀念投论文前的通宵工作，怀念您逐字逐句校正英文的辛劳，怀念论文拒稿时的沮丧，怀念老师们语重心长的开导。感谢两位老师不计较我的懵懂无知、任性拖沓，正是你们不辞辛劳的教导，才使我顺利度过了丰富多彩的研究生生活。

感谢计算软件与理论研究所的王大玲老师、林树宽老师、申德荣老师、杨晓春老师、朱静波老师、张天成老师、赵志滨老师、聂铁铮老师、董晓梅老师、张俐老师、胡明涵老师、吕鸣松老师、李芳芳老师、冷芳玲老师、李晓华老师、冯时老师、寇月老师，你们对学术的一丝不苟、对工作的认真负责、对生活的积极乐观，都为我树立了学习的榜样。尤其感谢林树宽老师关于并行程序课程的教诲引导我进入云计算的世界、吕鸣松老师关于人生道路的指引、李晓华老师关于答辩事宜的照顾和冷芳玲老师对我论文的仔细审阅与校正。

感谢 BSP 项目组的全体成员：郑虎、万强新、马越、周爽、刘金鹏、张楠、刘志诚、陈昌宁、杨宝兴、张天明、王倩、石佳莉。感谢每周六下午的讨论会让我们增加了彼此之间的了解。祝福 BSP 项目的全体成员在鲍玉斌老师的带领下不断取得战绩。

感谢师金钢、庞俊师兄和许嘉、王艳秋、李淼师姐不厌其烦的指出我研究中的缺失，在我迷惘时为我解惑。感谢同届的杨佳学、于洋、王宁、马茜、陈明涵和张慧、高春鹏、王文安、贾岩峰，我会永远记住与你们在一起奋斗的日子，很荣幸能与你们一起度过两年的学习生活。还要感谢下届的雷斌、刘璐、王彪、王璐璐，感谢与你们一起生活的高兴与快乐。

感谢我的室友以及班级同学，感谢你们对我的包容，与你们相识是我毕生的财富。

感谢我的父母，谢谢你们对我的理解和鼓励，因为有你们的支持，我才有勇气去面对困难。

最后，感谢所有关心帮助过我的人！

攻读硕士学位期间的论文项目情况

攻读硕士学位期间发表的论文：

1. **Zhigang Wang**, Yu Gu, Roger Zimmermann, Ge Yu. Shortest Path Computation over Disk-resident Large Graphs based on Extended Bulk Synchronous Parallel Methods[C], DASFAA, Wuhan, 2013, 7826(2): 1-15.
2. **Zhigang Wang**, Yubin Bao, Yu Gu, Fangling Leng, Ge Yu, et al. A BSP-based Parallel Iterative Processing System with Multiple Partition Strategies for Big Graphs[C], IEEE International Congress on Big Data, Santa Clara Marriott, 2013, unpublished.
3. Yubin Bao, **Zhigang Wang**, Yu Gu, Ge Yu, et al. BC-BSP: A BSP-based Parallel Iterative Processing System for Big Data on Cloud[C], DASFAA Workshops, Wuhan, 2013, unpublished.
4. Yubin Bao, **Zhigang Wang**, Qiushi Bai, Yu Gu, Ge Yu, et al. BC-BSP: A BSP-based System with Disk Cache for Large-Scale Graph Processing[C], The 7th OCSummit, Beijing, 2012.
5. Wenan Wang, Yu Gu, **Zhigang Wang**, Ge Yu. Parallel Triangle Counting over Large Graphs[C], DASFAA, Wuhan, 2013, 7826(2): 301-308.
6. 周爽，鲍玉斌，王志刚，冷芳玲，于戈. BHP: 面向 BSP 模型的负载均衡 Hash 数据划分[C], NDBC，哈尔滨, 2013，未出版.
7. 于戈，谷峪，鲍玉斌，王志刚. 云计算环境下的大规模图数据处理技术[J]，计算机学报, 2011, 34(10): 1753-1767.

攻读硕士学位期间参加的科研项目有：

1. 国家自然科学基金“云计算环境下的基于 BSP 模型的大规模图数据查询处理技术”（项目编号：61272179），2013-2016.
2. 国家自然科学基金重点项目分课题“数据密集型计算环境下的数据管理方法与技术”（项目编号：61033007），2011-2014.
3. 中央高校基本科研业务费项目“云环境下基于 BSP 的大规模磁盘驻留图数据处理技术研究”（项目编号：N110404006），2012-2013.
4. 教育部-中国移动科研基金“基于云计算的 BSP 并行计算方案研究与实现”（项目编号：MCM20125021），2013-2014.