# 大规模图数据的高效计算关键技术研究

（申请清华大学工学博士学位论文）

培 养 单 位：计 算 机 科 学 与 技 术 系
学　　科：计 算 机 科 学 与 技 术
研　究　生：章　明　星
指 导 教 师：武　永　卫　教　授

二〇一七年四月

# Research on Efficient Large-scale Graph Data Processing Technologies

Thesis Submitted to

**Tsinghua University**

in partial fulfillment of the requirement

for the professional degree of

**Doctor of Philosophy**

by

**ZHANG, Mingxing**

**( Computer Science and Technology )**

Thesis Supervisor : Professor WU, Yongwei

**April, 2017**

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

**（保密的论文在解密后应遵守此规定）**

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

# 摘 要

由于具有良好的表达能力，图数据结构被广泛用来对元素间具有复杂联系的数据进行建模，如社交网络、知识图谱等。而随着信息化技术的迭代更新和互联网应用的蓬勃发展，可以对大规模图数据进行分析的处理技术逐渐成为当前学术界和工业界的热门研究话题之一。已有为数众多的图计算系统被提出和应用，并取得了巨大的商业成功。

然而，现有的图计算系统主要基于一些简单化假设实现，如“点权不可分割”、“单个计算操作可以孤立地执行”等，因此很难达到下层硬件所能支持的最高计算效率。为解决这一问题，本文通过分析，发现大量图计算应用的主要效率瓶颈在于数据载入速度这一结论，并通过将不同环境下图计算系统的数据载入途径分为四个方面分别进行了研究。本文的主要创新成果包括：

（1）提出了一种三维图计算应用任务划分方法。该方法基于数据图中点权可进一步划分这一发现，拓展了一个全新的任务划分维度。与传统的一维和二维划分方法不同，三维划分方法允许将数据图中的点进一步地划分为子点并分配给不同的计算节点。测试结果显示，三维划分方法最高可以减少 90.6% 的通讯量，从而达成提升整体运行效率的目的。

（2）提出了一种分层的图数据组织格式。通过在外存设备上分层存储图数据，计算系统可以一次性载入更多的点，从而降低单个点重复读取的次数，达到提高计算效率的目的。测试表明基于这一设计实现的新型外存图计算系统较之已有系统有明显的性能提升，最高可达 6.4 倍的加速比。

（3）提出了一种矩阵图计算引擎的自动优化算法。该算法主要基于循环融合优化的原理，并同时考虑到了分布式环境下关于数据一致性的要求。在保证计算正确性的同时，该算法可以通过自动流水线化的方法提升图计算应用的数据局部性，从而减少内存带宽压力。实验表明该方法最高可将原程序加速 5.8 倍。

（4）提出了一种针对新型存算融合器件的图计算模型。针对存算融合这一全新的支持直接在内存器件上进行计算的体系结构，提出了与之相适配的新型图计算模型。该模型通过限定用户的编程接口，使得自动的通讯去冗余成为可能。进一步设计提出了基于广播树的更新传播算法，可以有效减少瓶颈链路上的通讯量。计算结果显示，上述两种方法最高可以减少近 20 倍的通讯量。

**关键词**：图计算；分布式计算；矩阵计算；局部性；PIM

# Abstract

Due to its good expressivity, graph has been widely used to model the relationship among data elements. As a result, many large-scale graph processing systems have been proposed and deployed in the real world, and have achieved great successes. The optimization for these systems is also a hot research topic in both the academic and industry.

However, the current implementation of graph processing systems are typically based on certain simplification assumptions, such as "vertex is indivisible" and "each compute operation can be executed isolatedly", and hence cannot achieve the best performance that the hardware can deliver. To resolve this problem, we investigate the field and found that the bottleneck of most graph applications is the speed of loading data. Based on this observation, we partition the load path of graph data into four stages and propose optimizations for them respectively. Specifically,

(1) We propose a novel 3D graph partition algorithm. This algorithm is based on the fact that the vertex property of many graph applications is actually divisible. Through exploring this novel dimension that have never been considered by existing methods, our algorithm can reduce up to 90.6% of the original communication cost.

(2) We propose a layered graph organization format. This format enables the processing system to load more vertices at a time, and hence reduce the average loading times of each vertex. As a result, our method can decrease total disk I/O for out-of-the-core graph processing systems and leads to up to a 6.4× speedup.

(3) We propose an automatic optimization algorithm for matrix execution engine. This algorithm is mainly based on loop fusion and has also considered the consistency requirement of a distributed environment. It is able to assure the correctness, and simultaneously, achieve a speedup up to 5.8×.

(3) We propose a process-in-memory-oriented graph processing framework. By enforcing certain constraints in the programming model, our framework makes it possible to automatically remove redundancy in the communication. It also provides a broadcast-based optimization that reduce communication load on bottleneck links. According to our calculation, it can reduce the communication cost to as low as only 5% of the original.

**Key words:** graph computing; distributed computing; matrix computing; locality; PIM

# 目 录

**第 1 章 引言** ..... 1
1.1 大规模图计算 ..... 1
1.2 图计算系统的分类 ..... 2
1.3 图数据高效计算的挑战 ..... 4
1.3.1 图计算的特点 ..... 5
1.3.2 现状和主要优化方向 ..... 6
1.4 本文的主要贡献 ..... 8
1.5 论文组织结构 ..... 10
**第 2 章 相关工作** ..... 11
2.1 基于分布式集群的图计算系统 ..... 11
2.1.1 分布式图计算中的基本概念 ..... 11
2.1.2 分布式图计算中任务的划分算法 ..... 13
2.2 基于外存的图计算系统 ..... 15
2.2.1 外存图计算系统的意义和挑战 ..... 15
2.2.2 以点为中心的外存图计算系统 ..... 17
2.2.3 以边为中心的外存图计算系统 ..... 18
2.3 基于矩阵的图计算引擎 ..... 20
2.4 基于存算融合硬件的图计算系统 ..... 24
**第 3 章 分布式图计算系统的三维任务划分** ..... 28
3.1 本章概述 ..... 28
3.2 实例研究：协同过滤问题 ..... 29
3.3 三维划分的基本概念 ..... 31
3.4 三维划分下的编程模型 ..... 33
3.4.1 数据模型 ..... 33
3.4.2 UPPS 下的三维划分 ..... 35
3.4.3 计算模型 ..... 36
3.4.4 二部图 ..... 37
3.4.5 与 GAS 模型的比较 ..... 37
3.4.6 例程 ..... 38
3.5 系统实现 ..... 42

3.5.1 数据载入和划分 ..... 42
3.5.2 Update 操作的实现 ..... 43
3.5.3 Push，Pull，和 Sink 操作的实现 ..... 44
3.5.4 基于矩阵的数据结构 ..... 44
3.6 实验结果 ..... 45
3.6.1 测试环境 ..... 45
3.6.2 微型测试集 ..... 47
3.6.3 实际应用 ..... 51
3.6.4 其他讨论 ..... 56
3.7 本章小结 ..... 60
**第 4 章 外存图计算系统的分层数据组织** ..... 61
4.1 本章概述 ..... 61
4.2 背景介绍 ..... 62
4.2.1 外存图计算系统中的一维划分：GraphChi ..... 62
4.2.2 外存图计算系统中的二维划分：GridGraph ..... 64
4.3 3DGridGraph ..... 66
4.3.1 分层存储优势 ..... 66
4.3.2 编程模型 ..... 68
4.3.3 实例研究 ..... 69
4.3.4 实现 ..... 69
4.4 测试结果 ..... 70
4.4.1 定量分析 ..... 70
4.4.2 实测结果 ..... 71
4.5 本章小结 ..... 73
**第 5 章 矩阵计算引擎的自动优化** ..... 74
5.1 本章概述 ..... 74
5.1.1 背景介绍 ..... 74
5.1.2 挑战 ..... 75
5.1.3 我们的贡献 ..... 76
5.2 设计思路与原理 ..... 76
5.3 KASEN 的编程模型 ..... 78
5.3.1 数据 ..... 78
5.3.2 数据的操作 ..... 79

5.3.3 实例：PageRank ..... 81
5.3.4 应用范围 ..... 82
5.4 KASEN 模型的具体实现 ..... 83
5.4.1 三种数据存储状态 ..... 83
5.4.2 显示的存储状态转化 ..... 85
5.4.3 约束条件 ..... 85
5.5 优化方法 ..... 87
5.5.1 循环融合 ..... 88
5.5.2 内部操作 ..... 89
5.5.3 DDAG ..... 90
5.5.4 优化算法 ..... 92
5.5.5 实例研究 ..... 94
5.6 估算公式 ..... 96
5.7 实现细节 ..... 97
5.8 性能测试 ..... 98
5.8.1 定量分析 ..... 98
5.8.2 性能测试 ..... 100
5.8.3 与已有系统的性能对比 ..... 101
5.9 本章小结 ..... 103
**第 6 章 拓扑感知的存算融合图计算方法** ..... 104
6.1 本章概述 ..... 104
6.2 背景介绍 ..... 106
6.2.1 互联拓扑结构 ..... 106
6.2.2 网络瓶颈 ..... 108
6.2.3 已有的 PIM 图计算系统 ..... 109
6.3 PGIM 系统 ..... 110
6.3.1 两阶段点程序 ..... 111
6.3.2 广播 ..... 114
6.3.3 计算和通讯的重合 ..... 116
6.4 测试 ..... 116
6.4.1 划分对通讯量的影响 ..... 116
6.4.2 广播对瓶颈链路通讯量的影响 ..... 117
6.5 本章小结 ..... 118

**第 7 章 总结与展望** ..... 119
7.1 论文主要工作总结 ..... 119
7.2 未来工作展望 ..... 120
**参考文献** ..... 121

# 第 1 章　引言

## 1.1　大规模图计算

随着计算机技术的不断发展，其应用范围逐渐推广到人类生产生活的方方面面。当前被计算机系统产生或者收集到的数据的规模正在飞速增长，这些大规模数据的存在无疑对现代计算机系统的大规模数据存储、管理、乃至于分析处理的能力提出了极其尖锐的挑战。为了应对这一挑战，以谷歌和微软为代表的各大公司以及诸多的学术界团队提出了一系列的大数据处理系统[1–5]。这些系统通过横向以及纵向扩展的方式极大地提升了数据处理系统的处理效率，扩展了它们所能承载的最大数据规模。

作为大数据中重要的一类，**大规模图数据**的管理与分析是近来备受重视的一个热点话题。举例而言，近来发展迅猛的各类社交网络就天然地产生了一个个规模巨大的图数据集。根据一项近期的统计结果[6]，Facebook 公司所维护的社交网络拥有将近二十亿的用户。如果以图的模型进行建模，其结果将会拥有近二十亿个点（代表各个用户）以及更高数量级的边（用于描述用户间的关系）。支持在这样规模的图上进行高效地查询和分析的需求无疑是现有系统所需应对的一项重要挑战。同时，由于图这一数据结构的灵活性，许多原本通过稀疏矩阵等其他数据结构进行建模的问题往往也都可以转化成图计算的形式。因此，正如我们将在后文第 3.2 节中详细描述的，诸如协同过滤等在内的很多机器学习和数据挖掘算法也都被认为是涵盖在图计算系统的处理范围之内的。这一现象极大地拓展了图计算的应用范围，因此也间接地提升了图计算系统的重要性。

鉴于大规模图数据的普遍存在以及对其进行分析处理的实际需求，众多商业公司和学术团队设计和提出了一系列的大规模图计算系统[1–4]。与传统的数据并行类大数据处理系统（如 MapReduce[7]，Spark[8] 等）不同，这些专用图计算系统专门针对图这一数据结构本身的特点进行了优化，从而避免了传统通用系统在图这一类数据间具有复杂联系的数据上可扩展性不好的问题。经过最近数年的发展，目前的大规模图计算系统已经具有了极高的可扩展性。根据 Facebook 公司研究人员的报告，他们通过将 Giraph[9] 系统横向扩展到数百台机器之上成功地对包含数十亿条边的超大规模图数据集进行了处理。与此同时，由于点程序（Vertex Program）等简单有效的编程接口的提出，程序员们可以非常简单地进行图计算的编程而无需考虑网络通讯、外存访问等复杂问题。往往系统可以几乎完全自动地将用户编写的单机单进程程序扩展到多核乃至多机环境下，甚至可以自动地进行单点容错。

由此可见，在可扩展性和易用性等方面目前已有的研究已经取得了非常良好的结果。也正是因为这样的原因，目前图计算系统已经被部署在众多的商业公司之中用于支撑它们的多种关键业务[6]。

## 1.2　图计算系统的分类

利用图对问题进行建模和计算是一门古老的数学分支，其历史最早可以追溯到欧拉于 1736 年对著名的哥尼斯堡七桥问题的论述，比第一台通用计算机的诞生时间还要早大约两百年。同时，由于图这一数据结构天然适合于研究用某种方式联系起来的若干事物之间的二元或多元关系，一经提出图计算就一直是热点研究方向之一。在计算机研究方面，各式的图论算法（如最短路径、最小生成树）和其它通过图进行建模的机器学习和数据挖掘算法（如 PageRank，聚落挖掘）都被广泛地应用于生产生活的方方面面，产生了巨大的价值[10]。然而，随着“大数据”时代的到来，目前被收集到并希望用计算机系统分析处理的图数据的规模正以极高的速度不断增长。如上一节中提到的 Facebook 社交网络的例子，目前需要计算的图数据的规模往往高达数百数千万、乃至上亿条边的级别。在这种情况下，传统的单机全内存图计算系统的计算性能和数据容量就变得不再能满足实际的需求。因此，目前有很多的新技术新方法被工业界和学术界的研究人员提出，用于搭建大规模图数据的处理系统。

具体来说，为了解决单机系统内存容量这一瓶颈所带来的问题，一个显而易见的方法就是开发**分布式图计算系统**。这些分布式图计算系统通过增加机器数的方式横向扩展系统的数据容量和计算能力，因此可以处理极大规模的图数据。前文中提到的 Giraph 系统就是一个典型的分布式图计算系统，其设计原理参考的是谷歌公司于 2010 年的 SIGMOD 会议上公开的 Pregel 系统的论文[2]。为了方便程序员的使用，这一类系统一般都会提供一个简单易用的上层编程抽象，也被称之为**编程模型**。基于谷歌公司提出的“像点一样思考（Think as a Vertex）”的原理，这些编程模型提供的接口函数一般仅仅允许程序员访问和操作整张数据图中一个点以及其相关的临域范围内①的数据。这样做的好处主要有两方面：（1）它极大地简化了程序员的编程过程。在利用这一编程模型进行编程的过程中程序员并不需要考虑各种各样的底层细节，如“当前被处理的这个点具体被分配到了哪台机器上”、“访问临域时相邻点的所在，是否需要进行远程通讯”等问题。因此，无论是单机还是分布式，也无论是一个包含多少台机器的集群，程序员完全可以用同样的一份程

① 在图中，一个点的所有入边和出边都被认为是在这个点的临域的范围之内的。同时，在部分的图计算系统之中，有的时候这些边相关的另外一个点也被认为是在临域与范围之内的。

序进行计算。系统的横向扩展基本上是完全由系统自动完成的。（2）通过限定每次只操作一个点的临域使得自动地并行化变得可能。自动并行之所以难以实现的最大问题在于很难自动地推断出一段程序会访问哪一部分的数据，通过直接在编程模型中对这一条件进行限定的方式，图计算的自动并行变得非常的简单："只要两个点的临域的交集为空，这两个点对应的程序就可以被并行地执行"。

虽然编程模型提供的抽象极大地方便了用户的使用，它同时也意味着用户无法直接访问到全局的信息，以至于很难进行特殊优化。因此，整个程序运行的性能大概率地取决于系统的实现。举例来说，由于用户编程时完全不需要考虑远程的通讯，因此整个程序最后产生的通讯量的大小就取决于系统对数据的划分这一自动的过程。如果系统选用了一个效果不佳的划分算法，就很可能导致通讯量过大，从而影响系统的整体性能[11]。事实上，由于图计算自身的性质，以及单机内存的带宽一般远大于网络带宽这一物理事实，网络通讯正成为制约当前分布式图计算系统效率的瓶颈之一。因此，设计分布式图计算系统时最重要的挑战之一就是如何设计一个好的任务以及数据划分算法。由于其重要性和实际价值，有一系列的工作都是针对这一问题的。相关的工作我们会在后文第 2.1.2 节中予以更进一步的介绍。

在分布式图计算系统这一方向之外，另外一个主要的大规模图计算系统类别为**外存图计算系统**。这一类系统通过将图数据存放在外存设备之上从而绕过单机内存容量的限制，使得无法访问集群的普通用户也能够处理相当大的规模的图数据[12]。

具体来说，这一类的系统会选择性的将部分当前需要的数据调入内存之中用于计算，同时将不再需要的部分数据换出，从而避免受限于单机内存容量的瓶颈。不过，为了保障这一计算过程的运行速度，外存图计算系统需要确保其对外存设备的访问具有良好的顺序性。众所周知，外存存储设备的随机访问带宽远低于其顺序访问带宽。然而，图计算本身的特性导致其对数据的访问具有极强的随机性。因此，如果简单地将点和边的数据顺序的存储在磁盘之上的话，最终系统的效率会明显地受限于极低的随机访问带宽。为了解决这一问题，目前的外存图计算系统都设计了精巧的数据组织格式或者访问模式[12–14]，从而极大地避免了对外存设备的随机访问。我们将会在后文第 2.2 节中具体地介绍这些技巧。总体来说，根据他们的测试，在中等规模的数据集下，基于相关优化工作的外存图计算系统可以达到由几十个节点组成的分布式图计算系统的效率。

除了上述的两种系统，单机全内存图计算系统[15–17] 依然是一个重要的研究方向。随着诸如三维堆叠[18]、PCM[19,20] 等存储技术的发展，在可见的未来内拥有 TB

级内存存储空间的机器也会越来越多，这一类系统的使用范围也因此会变的越来越广泛。更重要的是，单机全内存图计算系统所依赖的单机**图计算引擎**事实上是包括上述分布式和外存图计算系统这两类在内的所有图计算系统都共享的一个组件。即便是在分布式或者利用外存的情况下，系统某一时刻在某一特定机器上进行的计算依然可以看成是对一张子图进行单机全内存的计算，在原理上和直接在原图上进行计算没有什么本质上的区别。因此，在探索如何实现一个高效的图计算引擎的过程中发现和提出的技巧也可应用于分布式或者外存图计算系统的设计过程之中。具体来说，图计算引擎的优化主要分为两个方面：一个是如何利用单机多核架构进行多线程优化；另一个是如何优化程序的局部性，从而提升系统 cache 利用率。在本文中，我们主要关注的是后者的相关工作，其主要的方法在于利用图这一数据结构和稀疏矩阵在数学上的等价关系，把图计算问题转化成矩阵计算的问题，再利用高性能计算领域内已有的诸多矩阵计算优化技巧对其进行优化[5]。我们将会在后文第 2.3 节中进行更详细的介绍。

最后，与上述三种存储设备本身不具有计算能力的方案不同，近期还有一些**存算融合的图计算系统**被提出。这一类系统基于的是一种名为 Process-in-Memory（PIM）的新型的体系结构[21,22]。举例来说，Micron 公司主导提出的 Hybrid Memory Cube (HMC) 技术[23] 就是一种利用三维堆叠技术实现存算融合的方案。一般来说，一个 HMC 设备（一般被称之为一个 cube）会是由一组芯片组成的一个三维芯片栈。其中，每一个 cube 都是由多个存储层和一个逻辑计算层堆叠在一起组成的，因此这一类的设备不仅仅具有内存容量高的特点，还可以直接在内存附近利用逻辑计算层上的元件进行近内存的计算（Near-data computing）。由于可以被这些逻辑计算层利用的聚合内部带宽一般远高于外部带宽，利用上述的方法在数据传输到 CPU 之前进行预处理可以有效地提升系统的效率。不过，由于这一系统与传统的系统在体系结构上有着很多根本上的差别，已有的图计算系统往往都不能很好地利用它的特点，因此也无法发挥出很好的性能。为了解决这一问题，学术界的一些研究人员设计出了一些专门与这一新型体系结构的特点相配合的图计算系统[21,22]。具体的内容我们会在后文第 2.4 节中进行介绍。

## 1.3　图数据高效计算的挑战

早期的大规模图计算系统主要关注的是两个问题：（1）提升可扩展性。即通过将图计算系统横向扩展到单机多核乃至于多机分布式环境下获得更高的处理速度，同时通过增加计算节点的数量扩大系统能够处理的数据规模；以及（2）加强易用性。即通过设计简单的编程接口的方式抽象出一个编程模型，屏蔽底层复杂

细节方便程序员的使用。简单的说，这两个问题属于“能不能”的范畴之内。如果系统无法有效的扩展其计算速度和处理容量，那么它就无法应对当前海量数据所带来的挑战（要么直接存不下、要么存的下但是运算速度太慢了以至于没有实际意义）。同时，即便是解决了可扩展性的问题，如果系统的易用性不够以至于编程的难度高于普通程序员所能接受的范围的话，这样的系统即便存在也很难得到推广。相对的，由于在这两个方面取得的重大进展，目前大规模图计算系统在各方面都已经到达了可用的阶段，并且已经投入大规模实用，成为各大商业公司数据处理系统的标准配备之一[6,10]。

在解决了“能不能”的问题之后，自然而然的工业界和学术界的研究人员就将目光转向了“好不好”的方向。具体来说，当前图计算的主要热点研究方向在于回答当前的图计算系统的效率是否足够的高，是否完全利用了硬件的能力，还有没有优化的空间等问题。然而，由于图计算应用本身的一些特点，对其进行性能上地调优是非常困难的，因此早期的简单实现往往效率不高。

### 1.3.1　图计算的特点

具体来说，图计算应用主要有以下几个特点：

（1）其计算具有很高的访存计算比。拿 PageRank 这个最常见的图计算应用为例，在每一轮的迭代过程中，程序所需进行的计算不过是每条边做一次累和（计算新的 PageRank 值）、每个点做一次除法（除以对应点的出度）、然后再每条边做一次赋值而已，都是非常简单的操作。因此，在这种情况下程序的整体运行时间被数据的读写所主导。换句话说，对于一个单机的图计算应用，往往是内存带宽成为限制其速度的瓶颈而不是 CPU 的处理速度。

更糟糕的是，图计算应用还具有（2）数据局部性不好的特点，因此对系统 cache 的利用率往往都不高。具体来说，图计算典型的数据访问模式是在访问了图中的一个点之后访问其临域范围内的边和点。为了提升程序的数据局部性，一个简单的想法就是将一个点和其临域内的数据放在一起。但是，由于图中点与点之间复杂的联系关系，很难同时提升所有点的局部性。往往会出现的一种情况是提升了一个点的局部性的同时降低了多个其它点的局部性。所以，如何设计一个能够综合考虑全局的方法是非常重要的，但同时也是非常困难的。

在上述两点的基础上，图数据还具有（3）结构不规则的特点。和数据分布非常规则的稠密矩阵等数据类型不同，现实生活中的图数据往往分布非常不均匀。正如我们会在后文第 2.1.2 节中详细描述的，从实际生产生活中采集得到的图数据基本都符合 power law 的性质，即点的度数非常不均匀。可能有一小部分点拥有极其

大量的边与之相连，同时又有大量的点只有有限的几条边与之相连。在这种情况下，一个简单的规则的划分方法就经常产生非常大的负载不均衡，从而影响程序的效率[3]。

最后，图计算应用还有（4）受数据驱动的特点。基本上一个图计算应用在实际执行的过程中产生的访问模式等特性都取决于其输入的图数据。因此，即便是同一个种类的图计算应用，在接受不同的图数据作为其输入数据时表现出来的特性的差别也可能很大。而图数据本身的种类又非常丰富，这无疑增加了优化的难度。

事实上，上述的四个不同的特性是相互关联的。比如说（1）和（2），（2）和（3），还有（3）和（4）这几组特性之间都存在某种放大关系。正是因为图计算的访存计算比高所以对图计算应用来说其系统 cache 的利用率就显得非常重要。而图计算应用数据局部性不好的特性使得其对 cache 的利用率先天的就不算好。同时，由于图数据结构的不规则，对其进行局部性优化显然是比对稠密矩阵计算这一类规则数据上的优化要难得多。最后，图计算应用本身受数据驱动的特点导致用户也很难通过针对计算过程本身的特性进行效率优化，只能考虑针对图的特性。综上所述，由于图计算同时具有了上述的四个特性，其效率优化是非常复杂的一个问题。

### 1.3.2　现状和主要优化方向

鉴于图计算的这些特点，一个简单的实现和一个经过深思熟虑地优化后的实现相比经常会有非常巨大的性能差别。而在早期图计算系统的设计和实现之中，一般来说开发人员主要专注于解决之前提到的两个事关“能不能”的问题，比如说提升系统的可扩展性。相对的他们对单个计算节点上的运行效率的关注度不高，因此其硬件资源的利用也就都没有做到极限。举例来说，来自微软的研究人员在他们的调研过程中测量了一种被他们命名为 COST（Configuration that Outperforms a Single Thread）的指标[24]。一个系统的 COST 值为 c 意味着这一系统只有在横向扩展到 c 个计算节点的规模之后其运行速度才能超过一个经过仔细人工优化后的单线程实现。显然，COST 的值越小意味着系统对硬件性能的利用率越高。但是，根据微软的研究人员的测试，很多非常常用的分布式图计算系统的 COST 值都高达上百[24]。这也意味着，虽然这些系统具有不受单机内存容量限制、编程简单等诸多优势，但它们在运行效率方面都还有着非常大的优化空间。

为了解决这一问题，目前也已经有很多的工作着力于提升图计算系统的效率。相关的工作我们会在第 2 章予以分别介绍。不过总体上来说，由于对于图计算的

高访存计算比，对其性能的优化主要集中于提升将数据载入到合适的位置的速度。因此，根据不同情况下的数据的载入途径本文将优化的方法根据其应用场景分为四个不同的类别。

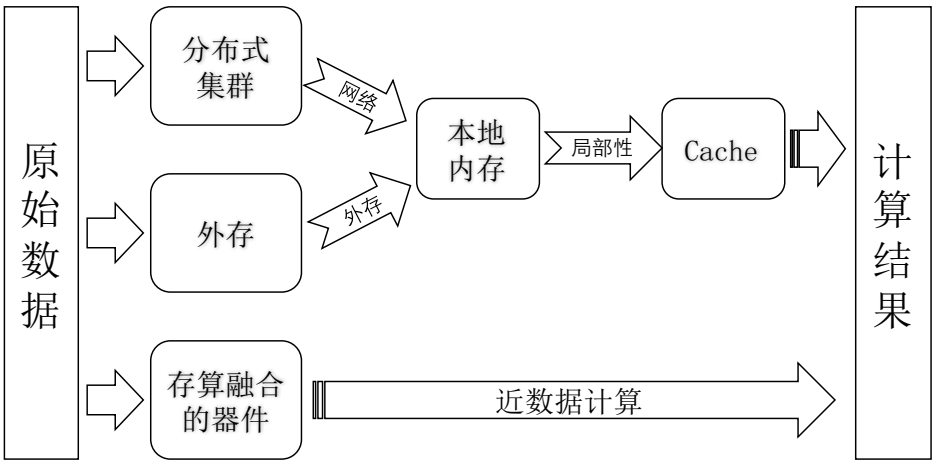



图 1.1　各种图计算系统中数据载入的途径

具体来说，图 1.1 给出了在不同种类的图计算系统下图数据的载入途径。首先，对于分布式图计算系统来说，其每一台单个的计算节点维护的是一张子图的数据。同时，由于数据图中各点之间普遍存在的联系，原图中部分点或者部分边会被同时指派给多个计算节点。在实际的结算过程中，这些同时被分配给多个计算节点的点和边会需要通过远程的数据通信来进行数据的同步。因此，对于这一系统，其数据的载入途径分为两个阶段：（1）一是从其它计算节点上通过网络将需要用于同步的数据读取到本地的内存之中去；然后（2）第二步则是和单机的图计算系统一样将数据载入到系统的 cache 之中以供计算。由于第二步从内存到 cache 是被多种不同类型的图计算系统所共享的，所以其相关的优化会在稍后提到。相对的，第一步的从网络上读取数据到本地的内存是分布式系统所独有的过程。为了尽可能地缩短这一步骤所需的时间，最显而易见的方法就是减少网络通讯的总量。而在分布式图计算系统中，对其运行时的通讯量起到决定性影响的因素就是其任务的划分方法。因此，目前有很多的工作着眼于如何巧妙地划分图计算任务和图数据，从而应对在保障各个计算节点间的负载基本均衡的情况下尽可能地减少通讯量这一问题。在后文的第 2.1.2 节中我们将对已有的几种划分方法进行介绍。总体来说，它们中很多都是基于图数据中点的度数分布是满足 power law 的这一性质进行的。

其次，我们在前文第 1.2 节中有提到，除分布式图计算系统之外，外存图计算系统也是很大的一类大规模图计算系统。在这一类图计算系统中，数据的载入过程也是分为两个步骤的，并且第二步和分布式图计算系统一样是从内存载入到

cache 中进行计算。不过，不一样的地方在于外存图计算系统数据载入的第一步是从外存设备而不是从网络将数据载入到内存中。因此，由于当前外存设备顺序访问和随机访问间巨大的带宽差距，在这一情况下主要的优化方法在于提升对外存设备访问的顺序性。我将在后文的第 2.2 节中简述几种图数据在磁盘上的组织和访问方式。它们应对这一挑战的主要方式都可以总结为“通过巧妙的数据组织，将随机访问转化成多次的连续块状访问”。

无论是从网络还是从外存设备，将图数据载入到本地的内存之后都需要进一步地将其载入到 cache 之中用于计算。而为了提升这一过程的效率，主要的方法就是提升程序的数据局部性，从而提升 cache 命中的概率。目前来说，这一环节主流优化方式是通过利用图这一数据结构和稀疏矩阵间数学上的等价特性将图计算操作转化成矩阵操作，再利用高性能计算领域内已有的针对矩阵计算的优化方法对其进行优化。具体的方法我们会在第 2.3 中予以介绍。

最后，与上述这几种路径都不同，目前以 PIM 为代表的新型体系结构给出了一种全新的基于存算融合技术的计算模式。存算融合的情况下，存储设备是和相应的计算设备用三维堆叠技术堆叠在一起的。换句话说，在这种体系结构中并没有多层的数据载入途径，而是直接在数据上进行计算。这一模式也被称之为“近数据计算”。不过，由于这是一种全新的体系结构，如果上层的系统不根据其独有的特性进行优化的话就没法发挥其最好的性能。因此，目前有一些研究人员提出了在这一架构下专用的图计算系统。后文的第 2.4 节将简述这一方向下的几种主要的方案。

## 1.4　本文的主要贡献

为了应对上一节中描述的四个方向上的挑战，工业界和学术界的研究人员已经提出了诸多的优化方法。它们分别可以用于缩短在图数据载入路径上的不同阶段的执行时间，从而在特定的情况下可以提升图计算系统整体的运行效率。然而，根据我们的调研发现，在所有的这四个方向上都依然还有可以优化的余地。因此，本文分别在这四个方向上进行了进一步的研究，提出了新的优化方法。经测试，通过将我们的方法和已有的优化方法组合在一起使用，我们重新实现的图计算系统能够拥有远高于已有系统的效率。具体来说，本文作出了如下的四个主要贡献。

（1）三维的图计算任务划分算法。

正如我们在前文中描述的，在分布式图计算系统中，通过减少通讯量缩短从网络将所需的数据载入到本地内存中这一步骤的时间是决定系统整体运行速度的一个非常关键的步骤。而对于分布式图计算系统来说，其远程通讯量的大小又主

要是由其所使用的图计算任务划分算法决定的。由于这一问题的重要性，其一直是一个热点的研究方向，也已经有许多的图计算任务划分算法被提出和实现。但是，经过我们的调研发现，这些已有的方法都假定图计算系统所处理的数据图中每一个点和边都是不可划分的单元。因此，它们都将图计算问题的任务划分等价成了图的划分问题。然而，在实际的应用场景中，有许多的数据挖掘和机器学习任务也被建模成图计算问题进行运算。在这种情况下数据图中经常会出现每一个点和每一条边的数据都是一个向量的情况，因此是可以被再次划分的。基于这一发现，本文提出了一种新型的划分方法以对这一之前未被重视的维度进行划分。换句话说，本文提出的新型划分方法将尝试将一张数据图中的每个点进一步的划分成子点，并把同一个点划分出的不同子点分配给不同的计算节点。这一维度在传统的一维和二维划分方法中都没有被考虑到，因此我们将这种划分方法称之为三维的划分方法。根据我们的实验结果，三维的划分方法可以极大地减少集群中各个计算节点之间的通讯量，从而提升系统整体的性能。

（2）分层的外存图计算数据组织。

与分布式图计算的情况类似，已有的外存图计算系统也都假定一张数据图中每个点的点权都是不可划分的单元。因此，在他们的数据组织格式中，一次载入必须完整的载入一个点所有的点权。在内存容量一定的情况下，这一限定条件就会限制一次性能够载入的点的数量的上限。而由于图计算的特殊性质，一般来说，一次能载入的点的数量越少也就意味着平均每一个点会被载入的次数越多。相对的，我们提出了在图计算应用允许的情况下将点权分层并分别存放的外存数据组织方法。通过将点权分层，我们提升了一次性能够载入的点的数量，从而减少了每一个点的平均载入次数。这也就意味着在点权方面我们减少了从外存设备到内存的载入数量，优化了数据载入路径上这一步骤的执行时间。但同时，我们的方法也意味着我们会增加每一条边的边权的平均载入次数。不过，根据我们的测试结果，通过选取一个合适的层数从而达成这一增一减之间的平衡，我们的新型数据组织格式可以减少外存读写的数量，从而提升系统整体的性能。

（3）基于矩阵的图计算引擎的自动优化。

前文中已经提到过，为了提升图计算引擎的效率，目前主流的一种方法就是通过利用图这一数据结构和稀疏矩阵间数学上的等价特性将图计算操作转化成矩阵操作，再利用高性能计算领域内已有的针对矩阵计算的优化方法对其进行优化。然而，根据我们的调研发现，目前基于矩阵的计算引擎大部分系统都采用了一种非常原始的运行方式，因此还有巨大的提升余地。具体来说，这些系统经常将每一个矩阵操作孤立开来单独执行。同时，程序的执行顺序也是严格地按照程序员

给定的顺序执行的。这就好像是单机程序在编译时完全关掉了编译优化，使用的是 ‘‘−O0” 选项进行编译的。虽然这种方法实现起来最为简单，但很明显还存在着极大地优化空间。为了解决这一问题，我们提出了一个基于数组的被称为 KASEN 的新型编程模型。KASEN 通过定义一个严格的计算和通信模型，使得分析程序并衡量他们的表现成为可能。基于这样的功能，我们设计了一个 KASEN 的自动优化器，可以自动对于程序员编写的原始程序进行高级优化。根据我们的评估，KASEN 的优化器可以显著地减少内存的读/写，缓冲区的内存开销和网络流量，从而达到了最高 5.82 倍的加速比。

（4）新型存算融合的图计算编程框架。

最后，为了应对新型存算融合体系结构所带来的挑战，我们提出了一种新型的图计算编程框架。与已有的针对存算融合体系结构的图计算系统不同，我们的系统同时考虑了（1）硬件的体系结构；（2）软件的运行环境；和（3）数据的划分和组织方式三方面的因素，并进行了协同地设计。事实上，虽然已有的系统中也有很多注意到了数据划分的重要性，不过在这些系统之中都是先决定好了系统的体系结构和编程框架之后再决定数据划分的方法的。然而，这一决定先天就使得很多有效的划分方法没有办法被使用。相对的，我们的新型系统（1）通过使用新的图数据划分算法极大的减少了跨 cube 通讯量，同时（2）在通讯时利用了广播的技巧去削减前文提到的瓶颈连接的通讯量，因此其对这些特殊的链接的压力也要小得多；最后（3）我们的编程模型还使得它可以非常简单的实现数据通讯和计算重叠的优化方法，从而进一步的隐藏了跨 cube 通讯的延时。

## 1.5　论文组织结构

本文余下的内容组织如下：第二章中，本文简述了已有大规模图计算系统的相关工作，并指出了其中的不足之处。在随后的第三到第六章中，本文分别描述了在四个不同的数据载入阶段上可以进一步进行的优化。依次为上一节中提到的“三维图划分算法”,“分层的图数据组织结构”,“图计算引擎自动优化器”，和“新型 PIM 图计算框架”。最后，在第七章中，本文简单的总结了全文的内容以及今后的研究方向。

# 第 2 章　相关工作

本章将按照前文第 1.2 节中给出的分类，分别介绍“分布式图计算系统”、“外存图计算系统”、“图计算引擎”、“存算融合的图计算系统”这四个类别下的相关工作。在介绍各个方向上已有研究的发展过程的同时，我们也会简略地提到他们的不足以及本文提出的新的方案的主要特点。

## 2.1　基于分布式集群的图计算系统

正如前文中所提到的，在大数据时代，用户所收集到并希望进行分析和处理的数据集的规模正以极快的速度进行增长。这一规律也体现在了图数据当中。举例来说，城市间的道路图是早期图计算系统所需处理的图数据中最大的一类[25]。不过即使是这一类的图数据一般也只是含有数万个点以及不到百万条的边，完全可以被单个计算节点容纳。但是，随着社交网络等大规模网络结构的发展，现有的图计算系统往往被要求能够高效地处理包含数以亿计条边的大规模图数据。很显然，由于单机容量的限制，这种大小的图数据无法被单机的内存完全容纳。因此，基于分布式集群的大规模图计算系统就逐渐成为一种主流的解决方案[6]。

### 2.1.1　分布式图计算中的基本概念

作为这一趋势的发端，谷歌公司在 2010 年的 SIGMOD 会议上公开的 Pregel 系统[2] 应该可以被认为是现代分布式图计算框架的起源。其定义的数据模型，即数据图（Data Graph）模型，和以点为中心的编程模型（Vertex-centric Programming）目前仍然是绝大多数后续系统参考和借鉴的对象。

具体来说，所谓数据模型定义的就是如何将一个具体的问题中所需要处理的数据进行形式化定义的方法。而在 Pregel 系统中所有需要处理的数据都必须要存放在如图 2.1 所示的在一张数据图里。依据定义，数据图为一张有向图，其在各个点之间以边表示的拓扑关系（Topology）的基础上还允许用户为每一个点或者边定义属性（Property），分别被称之为点权和边权。拿最常见的在网页互联关系图上求每一个页面的 PageRank 这一图计算应用作为例子的话，每一个页面会被表示成数据图中的一个点，而每一个从网页 a 指向网页 b 的超链接就表示成数据图中一条由 a 指向 b 的边。最后每个点上维护一个点权，即这一页面的 PageRank 值。可以注意到，在这个例子中边的权值是被省略了的。

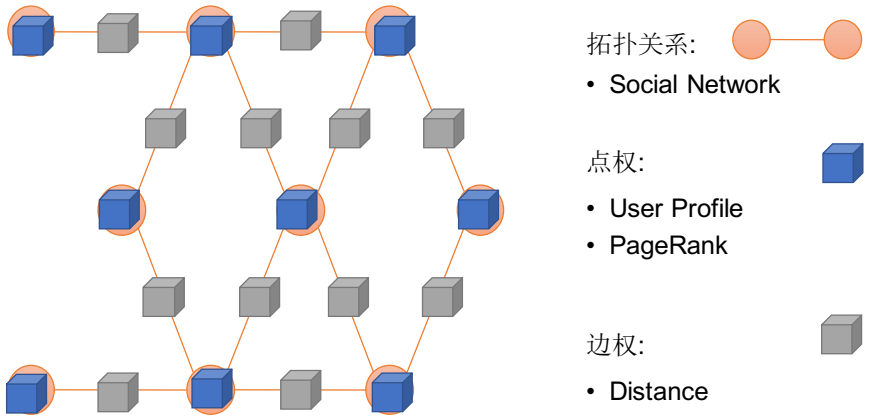



图 2.1　数据图模型。

在定义了数据之后，Pregel 系统的用户就可以通过 Pregel 提供的以点为中心的编程模型对数据进行操作。和数据并行型的编程框架 MapReduce[7] 一样，Pregel 系统的用户只需依据编程模型给出的接口进行上层逻辑的编写，而无需考虑底层的通讯等具体实现。在 Pregel 系统中，用户实现的程序被称之为点程序（Vertex Program），其操作范围仅为对应点的临域（即自己的点权和所有相连的边权），而且只被允许读取入边上的值和修改出边上的值。这些限制的主要目的在于方便并行的实现，理论上只要两个点对应的点程序没有操作同样的数据就可以并行地被执行。由于每一次操作的单元都是一个点，这一类的编程方法被又称之为按照“Think Like a Vertex”的思想进行设计的。

更进一步的，在用户对数据和计算都给予了明确的定义之后，Pregel 的计算系统将按照所谓 Bulk Synchronous Parallel（BSP）的计算模式[26] 自动地进行分布式地计算。BSP 模式的具体过程可以描述为“每一次计算由一系列的超步（superstep）组成，每一个超步又可以分成本地并发计算、全局通信和同步三个步骤；在本地计算时不产生任何通讯，相对的全局通讯时也不进行任何计算”。这种计算模式的好处在于其非常的简单，基本不需要做任何的并发控制，同时也便于实现基于检查点（checkpoint）的容错机制。同时 BSP 的缺点就在于同步的开销大，如果负载不均衡的话很容易得到次佳的执行效率。在 Pregel 系统中，每一个超步的本地并发计算阶段就是各个节点分别执行点程序的过程，而全局通信阶段则负责把产生的消息（附带在出边的边权上）送达相应的计算节点。由于 Pregel 采用的是基于点的图划分方法（即将数据图中的点均匀的划分给不同的机器，每个点与它所有的邻边都存储在一起），每一条被分割的边（即这条边的两个点被分到了不同的机器上）会产生一次远程通讯。

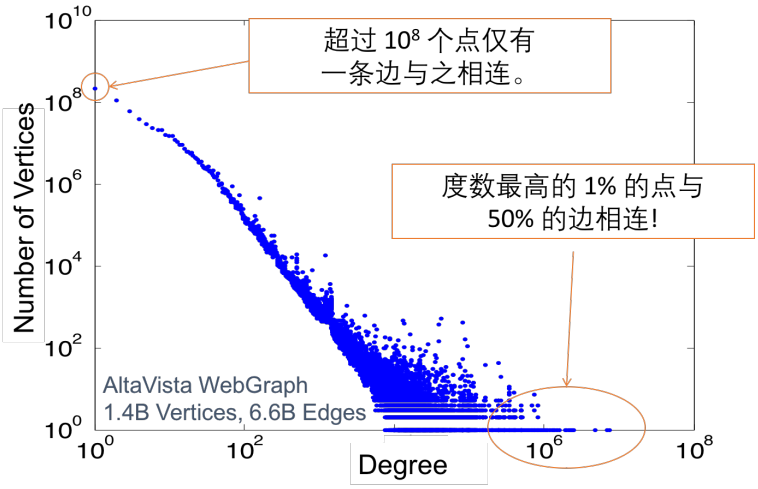



图 2.2　图数据中 Power Law 的性质[3]。

### 2.1.2　分布式图计算中任务的划分算法

在进行分布式图计算的过程中，一个非常重要的问题就在于如何对这个图计算任务进行划分，使得集群中的不同节点可以并行地进行计算。一个好的任务划分系统需要尽可能的同时做好两点：（1）一是避免负载的不均衡。可以想象，在上述的 BSP 模型下，每一个超步中的本地计算阶段都得等待所有节点中最慢的那个执行完毕才可以结束。这也就意味着如果任务的分配不够均匀，每一轮的超步中都会有大量的计算资源由于消耗在等待其他节点计算完毕的过程中而被浪费。这显然是不经济的；（2）二是尽可能地减少通讯的开销。虽然目前网络技术的发展也十分迅速，但不可否认的是在可以展望的未来中单机的计算速度远高于网络的传输速度的情况还是会普遍存在于实际环境之中的。因此，对于分布式计算系统来说，如何减少网络通讯，防止其变成阻碍系统横向扩展的瓶颈就会一直是一个重要的问题。而图计算由于具有计算一般比较简单的特殊性质，这一问题的重要性更加凸显。事实上，也有很多以调研结果表明通讯量的增加是影响图计算系统可扩展性的最重要的问题之一[11]。

由于图计算任务划分这一问题的重要性，目前已有很多工作对这一领域进行了研究。最早被广泛使用的方案就是上述第 2.1.1 节中提到的谷歌公司的 Pregel 系统所使用的以点为中心的划分方法，即将数据图中的点均匀的划分给不同的机器，每个点与它所有的邻边都存储在一起。在这种情况下如果一条边的两个点被分到了不同的机器上，它就会产生一次远程通讯。换句话说，在这种分割方式下，最终的通讯量是和被分割的边的数量成正比的。由于图和稀疏矩阵在数学上是等价的，而在表示成矩阵时一个点相关的所有边是被存放在同一行或者同一列内的，因此这种以点为中心的划分办法一般又被称之为一维（1D）的划分方法。

上述的一维划分方法由于其简单性被众多的系统所使用，而其效果在图中点所相邻的边数比较均匀的时候也还算可以接受。但问题在于，根据许多的调研结果，实际的图计算应用中常见的图数据往往满足一种被称之为 power law 的性质。如图 2.2 所示的 AltaVista WebGraph 数据集[27] 上的统计结果，在实际的图数据（real-world graph）中，经常存在一定数量的点占据了图中绝大数量的边的现象。换句话说，边的分布对于不同的点来说是非常不均匀的。在这种情况下对图进行简单的一维划分就会造成各个计算节点间重大的负载不均衡。正如我们在上文中所论述的，这会造成大量的计算资源浪费。

为了解决这一问题，卡内基梅隆大学的研究人员在 2012 年的 OSDI 会议上公开了一个名为 PowerGraph 的系统[3]。其中提出了基于边的划分方法，也称二维（2D）划分方法。简单地说，二维的划分方法就是将图中的边（而不是点）均分给各个计算节点从而达到负载均衡的目的。这是因为在大部分的图计算应用之中计算开销一般是和边数成正比的，因此在各个计算节点被分配的边数基本相同的情况下各个计算节点计算的负载就基本是均衡的。同时，由于那些度数非常高的点在二维划分的情况下可以被划分给多个节点，它们的计算就可以被并行化。为了达到上述的目的，最原始的点程序这一编程模型显然是不合适的，因为它要求一次就将与一个点相关的所有计算都完成。因此，PowerGraph 提出了一种与点程序类似，但在任务划分粒度上完全不同的一个计算模型 Gather-Apply-Scatter（GAS）。GAS 和点程序一样都满足每个点相关的计算只访问其邻域的局部特性，但 GAS 将点程序的执行过程明确地分割成了 Gather，Apply，和 Scatter 三个步骤，并通过强制要求 Gather 阶段对每一条边的结果进行聚合时的操作必须满足交换律和结合律的方式使得对一个点中不同边的计算的并行化成为现实。

如上所述，PowerGraph 系统与传统的 Pregel 系统最大的区别就在于它将拥有极大度数的那些点的计算也并行化了，从而降低了可能的负债不均衡。这解决了我们之前提到的任务划分的两个基本需求（即负载均衡和减少通讯量）中的第一个。不过，如果我们只是简单的利用边的哈希值将它们随机的划分给各个计算节点的话，依然有可能产生巨大的通讯量。具体来说，在二维划分方法中，如果与一个点相关的边被划分给了 X 个不同的节点，那么这个点在实际的系统中就需要维护包括一个主节点和 X-1 个副本在内的 X 份数据。同时，每一次 GAS 操作时主节点都必须收集相应的 X-1 个副本的中间计算结果并将最终计算后得到的值同步给它们。由此可见，在二维划分方法下通讯量的大小是和副本的数量成正比的。为了减少副本的数量，PowerGraph 以及随后的一些论文中都提出了各种各样的方法，然而它们经常难以在划分速度和划分结果这两个方面都达到一个很好的结果。在此基

础上，上海交通大学的研究人员在 2014 年的 EuroSys 会议上的论文提出了一个名为 PowerLyra 的系统[11]，其通过区别对待高度数点和低度数点的方法（hybrid-cut）达成了一个很好的结果，从而获得了那年 EuroSys 的最佳论文奖。

事实上，hybrid-cut 划分方法的原理其实很简单。当一条边的终点度数小于一个预先给定的阈值（一般设为 100 ～ 200）时 hybrid-cut 将按照这条边终点的哈希值对其分配，反之则按源点的哈希值分配。因为这样的划分方法基本保证了度数较小的点的所有边会被分配到同一个计算节点之上（相当于对这些点使用了一维的划分方法），同时又会将度数较大的点的边分配给不同的计算节点（即对度数较大的点使用的是二维划分方法），所以被称之为组合的（hybrid）划分（cut）方法。进行这样的组合划分的依据也很简单，可以概括为“度数大的点数量稀少且很难保证其副本的数量，因此我们干脆优先保证数量众多的度数小的点不存在副本”。

综上所述，经过长时间的研究，目前已有的划分方法在一般的图计算任务上已经能够同时在负载均衡和通讯量减少这两个目标上达成一个较优的结果。但同时，我们也可以看到，目前已有的工作都将图计算任务的划分这一问题等价成了图的划分问题。这一等价关系成立的原因在于目前已有的工作都假定数据图中各个点和边的数据都是不可再分的。然而，根据我们的调研发现，在一类重要的图计算应用中，其数据图中点和边的权值事实上是可以被再次划分的。依据这一发现，我们提出了一种新型的图计算任务划分算法并将其运用于分布式的图计算系统中，即三维的划分方法。实验表明，我们的方法可以极大地减少通讯量，从而提升了系统的效率。关于这一种新的划分方法，具体内容将在后文的第 3 章中进行详细的描述。

## 2.2　基于外存的图计算系统

### 2.2.1　外存图计算系统的意义和挑战

如上文所述，为了应对超出单机内存容量范围的大规模图数据所带来的挑战，通过设计支持横向扩展的分布式图计算系统，将图计算问题在分布式集群上进行并行运算是目前最主流的一个解决方案。这一方案也已经被包括 Google、Facebook 在内的各大公司所采用[6]。然而，分布式的系统自然而然的会引发诸多其他方面的问题。举例来说，在分布式系统环境下原本概率较低的机器故障问题出现的概率随着机器数的增加迅速增长，从而变得不可忽略。因此，分布式的系统大多都需要通过某种冗余的方式提供单点容错的机能。除此之外，分布式环境下计算环境十分复杂，很多时候即便用户通过选取合适的任务划分算法将先天的负载不均衡问题解决，依然有可能由于后天某些环境因素导致部分机器偶然的以较慢的速

度运行。对于这样的问题如果不加以处理的话就依然可能会产生大量的计算资源浪费。虽然这些问题也都有很多研究人员在进行研究，并提出了众多的办法，但相对来说都比较复杂难以运用。因此，不可否认的是，对于除各大公司以外的大量中小型用户来说，这些分布式系统自带的问题以及其他的比如维护调试难度等问题都使得分布式的图计算框架对于他们来说难以使用。

与此同时，来自卡内基梅隆大学的研究人员经过调研发现，对于绝大多数中小型的用户来说，他们所需处理的图数据一般大小最多不过是数十乃至百余个 GB 的级别[12]。虽然这个级别的数据量依然无法被常见的计算机的内存所容纳，但如果可以利用 HDD 或者 SSD 等外存设备的话，单机加以存储是可以达到的。因此，这些研究人员就提出了基于外存（out-of-core）的单机图计算系统。其愿景是希望能让普通用户也可以拥有处理大规模图数据的能力。

表 2.1　SSD 按照不同的连续块大小进行读取的带宽。

| 块大小 | 带宽 | 块大小 | 带宽 |
|---|---|---|---|
| 256MB | 767MB/s | 256KB | 470MB/s |
| 64MB | 772MB/s | 64KB | 202MB/s |
| 16MB | 728MB/s | 16KB | 69MB/s |
| 4MB | 662MB/s | 4KB | 20MB/s |
| 1MB | 596MB/s | 1KB | 6MB/s |

然而，虽然这一愿景十分的美好，但达成这一目的前提条件是需要能够设计出即使是使用了外存，也并不比只用内存的分布式文件系统慢上太多的计算系统。而为了设计出这样的系统，研究人员面临的最主要的问题就在于外存在顺序访问和随机访问间的速度差。基于磁盘的 HDD 技术在随机访问时速度极其低下这一点已经是众所周知的事实了。但事实上，即便是号称可以支持随机访问的 SSD 技术，其速度差依然是一个不小的问题。表 2.1 给出了我们所测量的一块 SSD 在以不同的大小为单元（block size）进行读写时 SSD 的吞吐量。具体来说，我们会随机从一个大文件中选一个位置作为初始位置，然后按照设定的块大小 B 连续读取 B byte 的数据，再随机选取另外一个初始位置进行读取，最终计算平均的带宽。从表中我们可以看到，这块 SSD 最高可以支持的读写吞吐为 772MB/s，但是一旦读写的单元大小低于 64MB 时，其吞吐量就随着单元大小的下降而迅速降低。当程序一次只读写大小为 1KB 的区块时，吞吐已经只剩下原来最高峰时的 0.7%，即 6MB/s。

更糟糕的是，图计算问题本身局部性并不好的特点极大地放大了上述吞吐速度差的问题。图 2.3 给出了两种最简单的图数据组织方式。可以看到，如果将图中的边以原点的顺序进行存放的话，虽然对原点的数据访问的顺序性很好，但是对

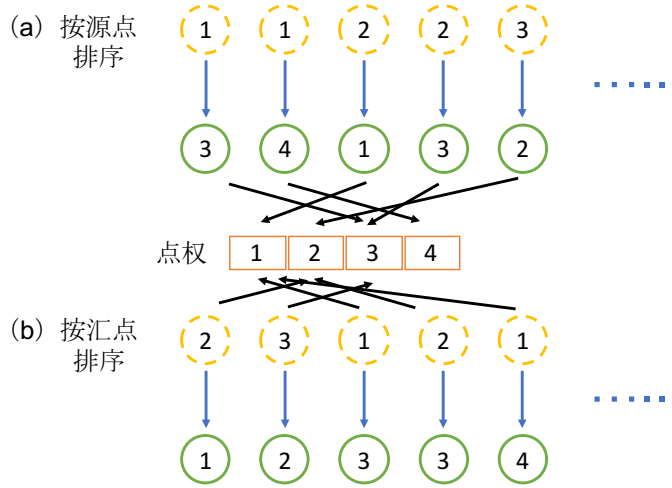



图 2.3　两种简单的数据组织格式。

汇点数据的访问就非常的随机以至于无法利用外存的带宽。相对的，如果将边按照汇点的顺序存放的话，虽然汇点访问的顺序性问题可以得到解决，但随之而来的就是对原点数据的访问变成了随机读写。因此，如何设计一种巧妙的数据组织模式使得对两种点的点权的访问都能够避免随机性是单机外存图计算系统首要考虑的目标。

### 2.2.2　以点为中心的外存图计算系统

为了解决上文中提到的随机访问的问题，来自卡内基梅隆大学的研究人员提出了一种被称之为 Parallel Sliding Windows 的数据组织格式，并将其运用到了他们提出的外存图计算系统 GraphChi 中[12]。具体来说，如图 2.4 所示，GraphChi 首先会把图中所有的点分为多个连续的 interval。比如说如果图中有编号为 1 ～ 10000 的一万个点的话，可以将其分为 1 ～ 100、101 ～ 300、301 ～ 400、401 ～ 1000 的四个 interval。可以注意到，GraphChi 并不要求每一个 interval 中所包含的点的数量是均等的。事实上对于 interval 的划分 GraphChi 只要求对于每一个 interval 中所含有的所有点来说，与它们相关的所有入边和出边的总和加起来是可以被单机的内存所容纳的。这一要求存在的主要原因在于 GraphChi 一次会计算一个 interval 内所有点的权值，因此需要能够将它们相关的所有信息（即这些点的领域）都载入到内存中去。这一条件的达成前提也很简单，只要图中最大的一个点其相关的所有边都能放在内存里的话就一定能找出这么一种合适的划分 interval 方法，因为每个 interval 最小可以只包含一个点。当然，在此基础上与每个 interval 相关的边数尽可能相等的划分方法自然也是最好的。划分完 interval 之后，边数据的组织方式同样是分块，只不过每一个块的名字不叫 interval 而是叫做 shard。在 GraphChi 系统中，每一个边的 shard 和一个点的 interval 是一一对应的，其存储的边的数据就

是这个 interval 中所有点的入边。也就是说，对于图中的一条边（s，t），如果点 t 是属于第 i 个点 interval，那么这条边一定是被分配到第 i 个边的 shard 中去的。在每个 shard 中，边的数据又是按照源点的大小进行排序的。举例来说对于 (1, 2) 和（3，2）这两个点，由于它们的汇点一样所以它们一定是被分配到同一个边的 shard 中的。同时，由于 (1, 2) 的源点 1 比较小，这条边的存储顺序比较靠前。

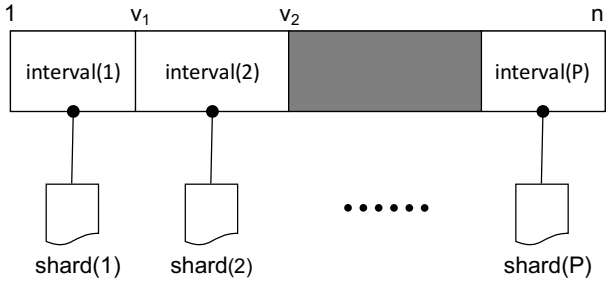



图 2.4　GraphChi 系统的数据组织格式[12]。

上述两种数据结构 interval 和 shard 定义了数据图中所有数据的存储方式。在划分完毕之后，GraphChi 的计算模式就是依次计算每个 interval 中点的数据。为了达成这个目的，在计算 interval i 中的所有点时需要载入相关的所有入边和出边。入边全部存储在第 i 个边的 shard 里所以可以一次性的全部读取进来。相对的，出边由于被分配在不同的 shard 之中所以需要分别读取。但是，由于在每个边的 shard 之中边的顺序都是按照源点进行排列的，所以可以想象，在每一个边的 shard 之中，其与某一个 interval 对应的出边的数据是连续存放在一起的，因此可以一次性读出而不用进行随机访问。由于数据的读取方式是入边数据一次读取每一个 shard，出边数据每次在所有的 shard 中从上到小读取连续的一块，因此每一次 interval 的计算都只进行了 interval 个数次的顺序读写，完全避免的小块的随机读写。根据 GraphChi 自己的测试，通过利用上述的数据结构其性能可以与一个大约有 100 个核的分布式计算集群相当。这显然是一个非常可观的数据。

### 2.2.3　以边为中心的外存图计算系统

GraphChi 通过巧妙地设计数据结构将图计算的运行过程中自然产生的大量随机读写统一成了每次 interval 计算若干次的形式，从而大大地提升了外存图计算系统的运行效率。但是，我们可以看到，对于一个被分成了 $P$ 个 interval 的图，在每一轮迭代的过程中每一个 interval 都依然会引起 N 次读写。也就是说总计每一轮迭代都会引起 $P^2$ 次读写。在 $P$ 的数额较大的时候这依然会是一个问题。因此，瑞士洛桑联邦理工学院的研究人员在 GraphChi 的基础上更进一步的提出了 X-Stream 系

统[13]。X-Stream 相较于 GraphChi 系统最大的区别在于其提供的是以边为中心的编程接口而不是以点为中心的，因此每一条边都可以独立进行运算。类似的改变我们在上述第 2.1.2 节介绍分布式图计算系统的发展过程中也可以看到，PowerGraph 系统提出的 GAS 编程模型就是以边为中心的，其替代的 Pregel 系统的点程序模型则是以点为中心的。

具体来说，X-Stream 提供的编程接口有两个分别是 Scatter 函数和 Gather 函数。其中 Scatter 函数的目的在于通过读取一条边（s,t）和其源点 s 的信息生成一个稍后会用来更新汇点 t 的更新信息。相对的，Gather 函数则是在拿到 Scatter 函数在边（s,t）上生成的更新信息之后将其作用于汇点 t 上。在用户通过这两个简单的接口定义了图计算的具体内容之后，X-Stream 的计算模式也非常简单。在计算的第一个阶段，X-Stream 会将所有的边以其按源点大小排序的顺序以流式的形式读入到内存里，并分别对其按用户定义的 Scatter 函数进行计算。由于边是按照源点大小的顺序进行排序的，可以想象进行 Scatter 函数的计算时对源点 s 的权值的访问也是非常有顺序性的，因此一直可以利用最佳的外存带宽。可以发现，这种模式和我们在图 2.3 中描述的那种简单模式（a），即把边按照源点排序一是一摸一样的。因此如果 X-Stream 系统希望能够在生成了一条边（s,t）上的更新信息后立即执行对应的 Gather 函数将其作用于汇点 t 之上的话就会面临非常严重的随机访问问题。不过 X-Stream 事实上并不会立即执行 Gather 函数，而是将 Scatter 函数生成的更新信息以数据流的形式顺序的记录到外存之上。只有当所有的边都遍历一遍之后，X-Stream 才会通过一次排序操作讲这些原本按照源点大小排序的更新信息重新排序转行成按照汇点排序的形式。在排序过后，X-Stream 就可以再一次的按照数据流的形式将更新信息依次读入到内存中并执行相应的 Gather 函数。可以看到，这个时候数据流的形式又变成了我们在图 2.3 中描述的那种简单模式（b），即把边按照汇点排序一是一摸一样的。由于 Gather 函数只会访问汇点的数据，所以再一次的是完美的顺序读写。

综上所述，X-Stream 本质上就是我们在图 2.3 中描述的两种简单模式的组合。其关键点就是在每一次计算时没有同时的访问一条边的源点和汇点，而是用一次外部排序将两个阶段连接起来。由于外部排序是一个已经被研究了几十年的经典领域，其效率很有保障。而之所以 X-Stream 能够做到这一点，最重要的就是它用以边为中心的编程模型替代了以点为中心的编程模型。此外，由于 X-Stream 每次访问的单元是一条边，而不像一个点可能有不定数条边相连，每一条边只和两个点相联。基于这一性质，在边权和点权都是固定大小的时候，我们可以很简单估算出每一次计算所需占用的内存大小。在这个能力的基础上，X-Stream 还设计了

一套可以最大限度地利用你系统 cache 的预取方案，从而进一步的提升了性能。根据 EPFL 的研究人员的测试，X-Stream 在实际应用中最高可以达到比 GraphChi 快上数倍的速度。

在 X-Stream 系统的基础上，后续还有一些其他的外存图计算系统被提出。比如清华大学的研究人员就在 2015 年的 ATC 会议上公开了一个名为 GridGraph 的系统[14]。GridGraph 系统采用的是和 X-Stream 系统的一样的以边为中心的编程模型。其主要特点在于将图数据以网格的方式分成了二维的区块。通过采取类似高性能矩阵运算中常用的 cache blocking 技巧的方式，GriadGraph 系统同时提升了对源点和汇点的反问局部性，从而达成了比 X-Stream 还要快上三倍左右的效率。由于本文即将在第 4 节描述的 3DGridGraph 就是基于 GridGraph 系统改造而成的，因此 GriadGraph 系统的设计细节会在第 4 节有更进一步的介绍，这里也就不重复叙述了。除 GridGraph 之外，同为 EPFL 的研究人员还曾经尝试将外存图计算系统和分布式图计算系统结合起来。他们提出的 Chaos 系统[28] 在每个计算节点上运行的是类似 X-Stream 的计算系统，但同时允许图数据被存储在多台机器的外存上。由于 Chaos 系统的具体细节与本文所要论述的系统并不相关，因此这里也不再赘述。

## 2.3　基于矩阵的图计算引擎

上述两节分别描述了在分布式集群和单机外存这两个不同的环境下当前已有的图计算系统的演化路径。不过，虽然环境不同，但核心上这两类问题都是在解决同样的一个问题，即如何尽可能快的将图计算应用运行时所需要的所有数据载入到一台机器的内存中去。只不过在外存环境下数据是从外存读取的，而分布式环境下数据则有可能需要通过网络通讯从其它机器中读取。由于无论是外存还是网络，其速度往往无法与计算相比拟，因此如果这一类的问题没有得到妥善处理的话，它们就会成为系统性能的瓶颈，使得优化系统的其他部分都不会产生明显的效果。不过，如上所述，在研究人员的努力下诸多巧妙的方法被提出，极大的提升了数据读取的速度。同时，随着存储以及网络技术的不断发展，特别是进行 NVMe SSD 和 InfiniBand 等技术的成熟，外部数据输入的带宽与内部内存的带宽之间的间隔正在不断地减少。在这样的环境下，如何通过提升系统 cache 的利用率等方法加速内存中图计算应用的计算速度这一话题的重要性也与日剧增。因此，如何设计高效的图计算引擎这一话题近期也吸引了众多研究人员的兴趣[5]。

在最早的时候，图计算系统的计算引擎都是简单的按照点为中心设计的。具体来说包括每次计算一个点对应的点程序，通过哈希表等方式进行点权值的寻访等等。由于图计算应用本身局部性不好的特点，这样的图计算引擎对 cache 的利用

率极低，因此性能不高。事实上，这一问题的本质和之前在第 2.2 节中描述的外存图计算系统的问题是一样的。只不过因为原来系统性能的瓶颈并不在计算上所以才用了这样一种实现起来比较方便的方法。相对的，在系统 cache 利用率的重要性与日剧增的当下，这一简单实现方法显然变得不再可取起来。

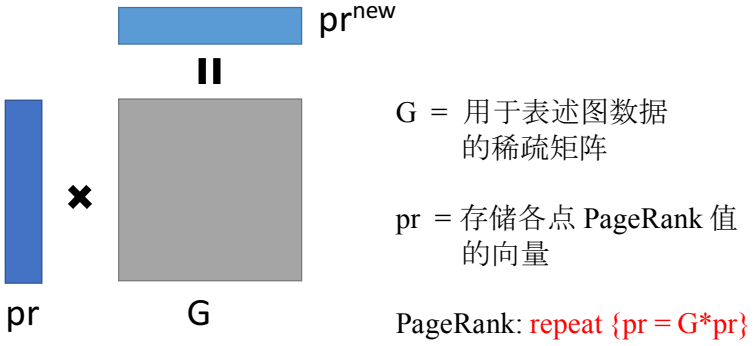



图 2.5　PageRank 问题的矩阵表示形式。

为了解决这一问题，伯克利的研究人员提出了利用图数据和稀疏矩阵数据在逻辑上是等价的这一特性对图计算进行优化[5]。举例来说，图 2.5 描述了一个非常常见的图计算的例子 PageRank 以及其对应的矩阵计算模式。在图计算模式下，求取一张图上每个点的 PageRank 值意味着：在每一轮迭代的过程中，每一个点将其由上一轮迭代计算得到的 PageRank 值平均的分配给其相关的每一条出边所指向的点。而在矩阵运算的模式下，数据图中所有点的点权被建模成一个稠密的数组 pr。数组 pr 中的第 i 个元素存储的就是原图中第 i 个点的权值。相应的，数据图中边的数据被建模成一个稀疏矩阵 G 进行存储。稀疏矩阵中的每一个非零元（i,j）都对应于原图中从第 i 个点指向第 j 个点的一条边，而这个非零元的权值则是对应源点 i 度数的倒数。在这种建模方式下，PageRank 算法中每一轮迭代的计算过程就可以简单的描述为求稠密向量 pr 和稀疏矩阵 G 的乘积。

针对图和稀疏矩阵之间这样的等价关系，伯克利的研究人员通过调研找到了一系列可以等价于稀疏矩阵运算的常见的图计算操作，并将它们集合形成一个以矩阵为中心的图计算框架 CombBLAS[5]。CombBLAS 提供了一系列的矩阵计算操作，其中就包括上面 PageRank 例子中用到了的通用稀疏矩阵向量乘法（Generalized Sparse-Matrix-Vector Mutiplication, GeSpMV）。根据相关的调研，CombBLAS 可以被用来实现许多重要的图计算应用。

与此同时，由于矩阵计算是一项已经在高性能计算领域被研究了数十年的领域，CombBLAS 在实现的过程中借用了很多已有的优化技巧，从而达成了很高的效率。举例来说，以何种的顺序对稀疏矩阵中的非零元进行存放是一个重要的问

题，其质量的优劣严重地影响计算的局部性或者说系统对 cache 的利用率。与图 2.3 中描述的那两种简单的图数据存储方式类似，对于稀疏矩阵也有两种简单的存放方式。其中对应于图 2.3 中按照源点大小顺序进行存放的方式是按照稀疏矩阵的非零元的行号大小的顺序进行存放，这种方式一般被称之为 row-major。同样的，对应于图 2.3 中按照汇点大小顺序进行存放的方式是按照稀疏矩阵的非零元的列号大小的顺序进行存放，也称之为 column-major。同样，按照 row-major 方式进行存放的话对于行数据的访问局部性好但是列数据访问的局部性差，而 column-major 则是反过来。为了能够同时优化行列双方的局部性，有诸多的技术被提出。其中就包括上文第 2.2.3 节中提到的 cache blocking 技术。除此之外，被基于矩阵的图计算引擎利用的一种顺序按照希尔伯特曲线的顺序进行存储，也称之为希尔伯特顺序（Hilbert Order）。由于希尔伯特曲线的具体定义涉及复杂的数学原理，这里不展开叙述。简单的说，希尔伯特曲线是一根能够不重叠的遍历格状大正方形中所有小正方形格子的连续曲线。不过，除了连续和不重合这两个特性之外，希尔伯特曲线的另外一个特性就在于其倾向于连续访问两个在二维空间上距离比较近的两个格子（如图 2.6 所示）。因此通过将稀疏矩阵中的非零元按照被希尔伯特曲线经过的先后顺序进行排序的方式可以较好的同时提升行数据和列数据的局部性。

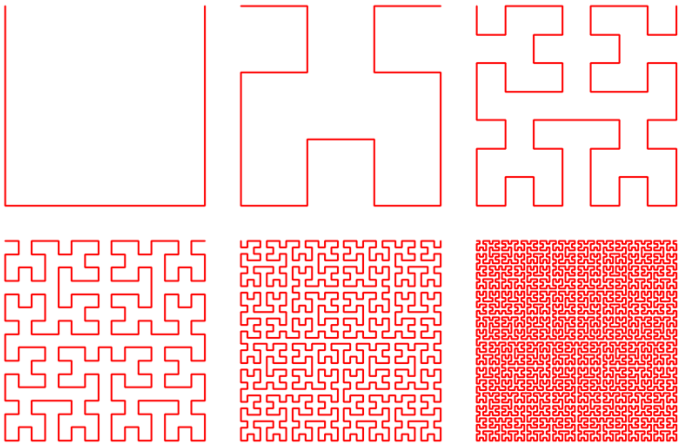


图 2.6　各种矩形边长下的希尔伯特曲线[29]。

综上所述，基于矩阵的图计算引擎通过利用高性能计算领域中已有的优化方法能够达成比传统的简单图计算引擎更高的系统 cache 利用率，从而达成了更好的效率。不过，不可否认的是，以矩阵为核心的编程接口较之传统以图中点或者边为核心的编程接口在使用时对程序员的负担更大。同时，以图中元素为核心的编程接口已经被广泛地采用，因此会产生一定的迁移开销。为了解决这一问题，英特尔的研究人员设计了一个名为 GraphMat 的图计算系统[15]。该系统最大的特点

在于它在提供了一个以点为中心的编程模型的同时，自动地将用户编写的程序映射成矩阵的操作并交给一个矩阵计算引擎进行计算。这一自动映射的方法在给予用户方便的编程接口的同时达成了高效的后台计算。由于这一方式的优越性，事实上本文提出的计算系统 CUBE 也是采用的这一种模式。只不过 CUBE 对其进行了进一步的扩展，将 GraphMat 的单机计算引擎扩展成了分布式的计算引擎。

由于矩阵计算在效率上的明显优势，以及其在除图计算以外其他的数值分析领域（比如数据挖掘与机器学习）上的良好表达能力。除去上述的基于矩阵的图计算系统之外，还有一系列的专门针对矩阵计算的计算框架。近期比较著名的有由谷歌公司提出并开源的 TensorFlow 系统[30]。在 TensorFlow 系统中，用户的程序是由一个有向无环图（DAG）描述的。图中每一个点表示一个矩阵的操作，而点之间的每一条边都是一个 N 维的数组，也被称之为 Tensor。程序实际的运算过程就是沿着 DAG 的拓扑顺序执行每一个操作，操作的入边表示输入顺序而出边表示输出数据。同样的架构还被用于许多其他的系统之中，比如说惠普公司在 2014 年的 EuroSys 会议上公开的分布式的 R 语言实现 Presto[31]。

和上文中提到的基于矩阵的图计算框架一样，这些基于矩阵的分布式计算框架通过借助于高效的矩阵操作达成了非常高的效率。根据他们的实验结果，一般来说同样的一个机器学习程序在他们这些基于矩阵的框架上的实现能够比传统的基于 MapReduce 或者说 Spark 的实现快上一个数量级。

然而，经过我们的调研发现，虽然这些系统较之传统的方法已经有了很大的提升，但依然还有很大的提升空间。之所以可以做出这样的论断的原因在于目前的基于矩阵的计算框架都采取了一种比较简单的执行方案。具体来说，这些系统一般来说都将每一个矩阵操作孤立开来单独执行。同时，程序的执行顺序也是严格地按照程序员给定的顺序执行的。打一个比方的话，就好像是单机程序在编译时完全关掉了编译优化，使用的是“-O0”选项进行编译的。虽然这种方法实现起来最为简单，但很明显还存在着极大地优化空间。

举例来说，由纽约大学的研究人员在 2015 年的 ATC 会议上公开的 Spartan 系统[32] 是一个分布式的 NumPy 实现。其中就提供了一些简单的优化方法，比如说将两个连续的 Map 操作组合成一个。即便是只提供了这样一些简单的操作，根据他们的实验结果，也仍然可以产生上至三倍的效率提升。可以想象，如果有一个更加通用的优化器的存在的话所能达到的效率提升肯定会更加可观。本文第 5 章所描述的就是这么一个通用的分布式矩阵计算框架优化器。

## 2.4　基于存算融合硬件的图计算系统

在第 2.3 节的开头有提到，虽然环境不同但事实上无论是分布式图计算系统还是外存图计算系统，他们最重要的优化点都集中在如何尽可能快的将计算所需的数据载入到单个计算节点的内存之中去这个过程上。只不过在分布式环境下是从网络载入，而外存环境下是从外存载入。用同样的视角去看的话，上一节中提到的计算引擎的优化其实也可以归纳成同样的类别，只不过在存储的层级上更高一级而已。简单的说，计算引擎的优化基本集中在如何提升 cache 利用率，也就是如何尽可能快的将计算所需的数据载入到 cache 之中。这显然是和当前计算机的分层存储架构，即 cache、内存、外存/网络这样按照速度上下分层的结构是吻合的。这种架构十分有效的提供了一种让运算速度尽可能接近上层存储但数据容量又可以接近下层存储的方法。但为了利用这样的架构，程序需要具有较好的局部性，不然的话就会由于系统需要频繁的访问下层存储而使得上层的缓存不起作用。可是，由于图计算本身的特点，图计算应用的局部性一般都不太好，这也就是为什么会有这么多的工作希望能够通过在这一方面的优化提升图计算的性能的原因。不过，不可否认的是，虽然已有的工作在最初的基础上已经取得了长足的进展，但依然无法完全解决这一问题。因此，在新型的支持存算融合的硬件出现之后，就有很多的研究人员寄希望于它们可以一劳永逸的解决这一问题。

事实上，早在十九世纪九十年代就有一些研究人员提出可以通过给存储器件加上计算能力来减少外存与内存之间的数据交换，进而达到绕过带宽的瓶颈的目的[33–35]。然而，这些早期的尝试都没有取得很大的成功。回过头来总结的话，这一失败主要是有两方面的原因。一是上述给存储器件添加计算能力的工艺较为复杂难以实用；二是当时并没有找到特别合适的应用领域，以至于难以推动工业界跟进研究。但是，随着所谓大数据时代的来临以及三维堆叠技术的发展，这两个主要的问题都一定程度的得到了解决。这也是为什么近期有大量的研究人员重拾了对存算融合体系结构的兴趣。

具体来说，在软件应用领域方面，本文的主题图计算就是一个非常好的目标领域。在图计算当中，操作图中的一个点往往意味着要访问这个点相邻的所有点，这使得图计算程序的局部性一般都很差。正如前几节所列举的，对于这一类的应用，如何提升数据的加载是非常重要的问题。同时，在硬件方面，三维堆叠技术的发展也给实现支持存算融合体系结构的硬件设备提供了基础。举例来说，Micron 公司主导提出的 Hybrid Memory Cube (HMC) 技术就是目前最为现实的一种实现方案[23]。如图 2.7 所示，一个 HMC 设备（由于及结构一般也被称之为一个 cube）是由一组芯片组成的一个三维芯片栈。每一个 cube 都是由多个存储层（memory

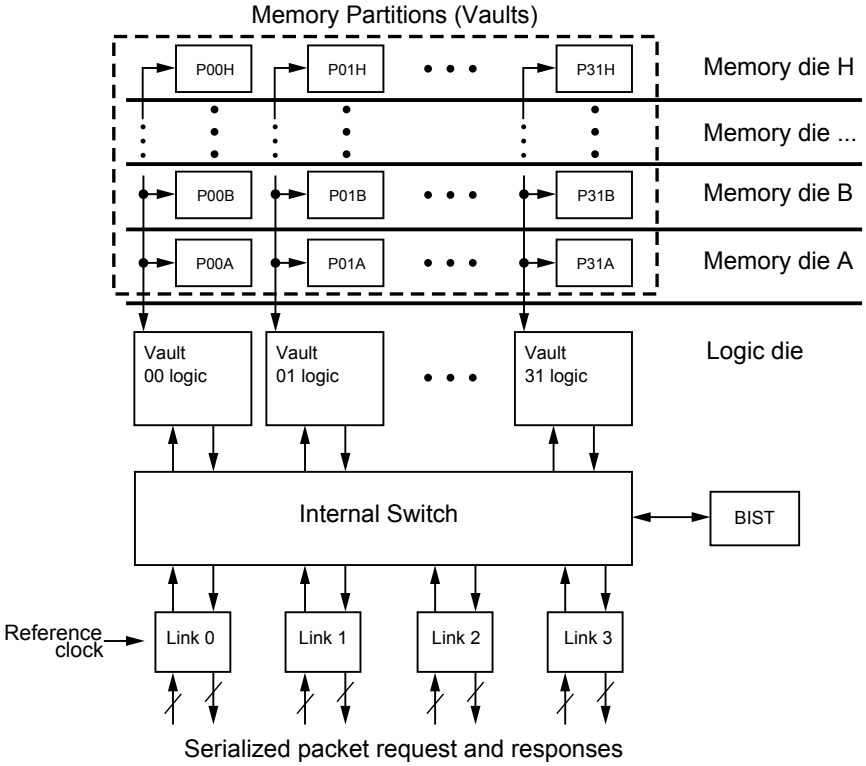



图 2.7　HMC 的内部结构[36]。

dies/layers）和一个逻辑计算层（logic die/layer）堆叠在一起组成的。

在这样的体系结构下，一共有两种不同的带宽：（1）一种是内部带宽，其限制的是在同一个 cube 内逻辑计算层访问本地存储层中的数据的最大速度；（2）另一种是外部带宽，其限制的是 cube 提供给外部器件（比如说其他的 cube 或者宿主 CPU）进行访问的带宽。图 2.7 给出的就是一个实际的 HMC 的例子，其配置与 Hybrid Memory Cube Specification 2.1[37] 中给出的相同。从图中我们可以看到，一个 HMC 的 cube 中包含 32 个纵向的切片，其中每一个切片一般被称之为一个 vault。除此之外，一个标准的 HMC 的 cube 还包含 4 根双路外部链接线（serial links）和一个将这些部件连接起来的内部交换网络（crossbar network）。在每一个 vault 之中又总想分布着一个逻辑计算层和多个存储层。根据标准，一个 vault 中的存储层最多可以存放 256 MB 的数据。也就是说，每一个 HMC 的 cube 最多可以存放 32*256MB = 8GB 的数据。在一个 vault 中的各个层之间一种低能耗的基于 TSV 技术的通路被用于连接逻辑计算层和数据存储层。由于每一条 TSV 通路可以提供上限 10GB/s 的带宽，一个 HMC 的 cube 内部的访问带宽最高可达 32*10GB = 320GB/s。与之相对的是，在标准配置下 cube 的每一条双路外部链接线只包含 16 条入线（input lanes）和 16 条出线（output lanes），总共能提供 480GB/s 的外部带宽（即每条外部链接线 120GB/s）。虽然从这个数据来看似乎外部带宽比内部带宽还要大，但事实上由于内部带宽是可以横向扩展的，在实际系统中 cube 的外部

带宽会成为影响程序效率的瓶颈。换句话说，如果一个计算系统由 N 个 HMC 的 cube 构成，那么在理想情况下聚合的内部带宽可以高达 N*320GB/s。相对的由于互联的原因，很可能总系统的外部带宽依然是只有 480GB/s 甚至可能更低。相关的详细情况后文第 6 章中有更进一步的描述，因此这里就不重复说明。当然，无论如何，由于基于 HMC 的架构可以支持内部聚合带宽的横向扩展，它也被称之为具有提供与内存容量成正比的带宽的能力（memory-capacity-proportional bandwidth）。为了利用这一能力，HMC 提供了将一定的计算功能整合到各个 cube 的逻辑计算层的能力。理论上来说，每一个 cube 中的各个 vault 的逻辑计算层都可以被看成是一个小型的通用 CPU。只不过由于片上面积及能耗等方面的限制一般来说这些小型 CPU 的功能多会设计的比较简单，比如说是 single-issue 和 in-order 的。同时，除了通用 CPU 之外，这些逻辑计算层之中也可以集成一些简单的专用加速器，以提升系统的性能。

在同时拥有了可以适用的软件应用范围和可以实用的硬件实现技术之后，正如我们前面所提到的，很多研究人员重新燃起了设计存算融合的体系结构的兴趣。他们提出了一系列的架构设计方案以及相应的软件编程框架。其中，与本文第 6 章所要描述的系统最相关的是由卡内基梅隆大学的研究人员在 2014 年的 MICRO 会议上公开的 Tesseract 系统[21]。Tesseract 是一个基于 HMC 技术的大规模并行图计算框架。它提供的是一个类似于谷歌公司的 Pregel 系统的以点为中心的编程模型，因此程序员可以很方便地在其之上进行应用的编程而不需要直接操作 HMC 的硬件。这极大地方便了程序员的使用和降低了 HMC 技术的普及难度。根据他们的测试，由于利用了多个 HMC cube 系统的聚合内部带宽，Tesseract 较之已有的单机内存图计算系统最多可以快上整整一个数量级。然而，虽然 Tesseract 和以前的没有利用 HMC 结构的系统比较的话已经很快了，但事实上 Tesseract 依然没有很好的利用的多个 HMC cube 系统的极大的聚合内部带宽。它们自己的测试结果显示，Tesseract 对内部聚合带宽的利用率一般不超过 40%。根据我们的调研，这种现象的主要原因在于 Tesseract 系统大量引发了跨 cube 的外部通讯。正如我们上面提到了，这些外部通讯只能依赖无法横向扩展的外部带宽完成，因此很容易就成为了瓶颈。更糟糕的是，为了互联多个 HMC 的 cube，Tesseract 系统采用的是一种被称之为 Dragonfly 的互联拓扑结构。虽然 Dragonfly 较之简单的托普互联结构（比如 Mesh）有这直径短、内部联通度高等特点，它依然不是一个全对称的结构。这就使得在互联网络中存在一些连接可能会需要承受比其他连接高的多的负载。这种不均衡性更进一步的使得外部带宽不够的问题放大，影响了整体系统的性能。而本文中第 6 章所描述的系统就是为了解决这一问题设计的。它考虑到了拓扑的实

际情况选取了合适的通讯模式，从而减少了对外部带宽的依赖，进而取得了更好的效率。

# 第 3 章　分布式图计算系统的三维任务划分

## 3.1　本章概述

正如前文第 2.1.2 节中所强调的，高效的分布式图计算系统依赖于其任务划分组件的良好运作。这是因为任务划分质量的好坏不仅直接决定了系统中各个计算节点间负载是否均衡，也是决定通讯量是否会成为影响性能瓶颈的关键因素。然而，虽然已有许多的工作提出了各种各样的划分算法，这些已有的方法都认为图计算系统所处理的数据图中每一个点和边的权值都是不可划分的单元[3,4,11,38]。因此，它们都将图计算问题的任务划分等价成了图的划分问题。可是，在实际的应用场景中，有许多的数据挖掘和机器学习任务也被建模成图计算问题进行运算[3,4]。在这种情况下，数据图中每一个点和每一条边的权值经常是一个向量，因此是可以被再次划分的。

基于这一发现，本文提出了一种新型的划分方法以对这一之前未被重视的维度进行划分。换句话说，本文提出的新型划分方法将尝试将一张数据图中的每个点进一步的划分成子点并把同一个点划分出的不同子点分配给不同的计算节点。这一维度在传统的一维和二维划分方法中都没有被考虑到，因此我们将我们新提出的这种划分方法称之为**三维的划分方法**。根据我们的调研，三维的划分方法可以极大地减少集群中各个计算节点之间的通讯量，从而提升系统整体的性能。

定性的说，三维划分和已有划分算法的主要区别在于：每一个子点都只包含原来的完整点权中一部分的数据，因此我们可以将更多的子点放在同一个计算节点之中。在原来的情况下，由于单机内存容量的限制，这些点只能被分配到更多的计算节点之上。因此，在好的一方面，原来需要通过跨节点通讯才能互相交流的一些点被分配到了同一台机器上，因此可以用更为廉价的本地通讯进行信息的交互。但是同时，在坏的一方面，三维划分引发了一种原来并不存在的通讯方式，即同一个图中的点的不同子点之间的通讯。不过，无论如何，我们通过提出三维划分给予了用户更大的选择范围。我们的用户可以自由的选择是否进行三维的划分以及在第三维上进行多少层的划分。根据我们的调研发现，通过选择最优的三维划分配置，我们的划分方法较之传统的一维和二维划分方法可以极大地减少通讯量。同时，我们的研究还发现，三维划分并不仅仅对点权向量大小很大的场景有效，即使是在点权向量的大小比较小的时候也是很有效的。举例而言，我们的实验结果显示我们仅需在第三维上划分出 64 层就可以减少最多 90.6% 的通讯量。这只需中等大小（>64）的点权向量就可以做到。

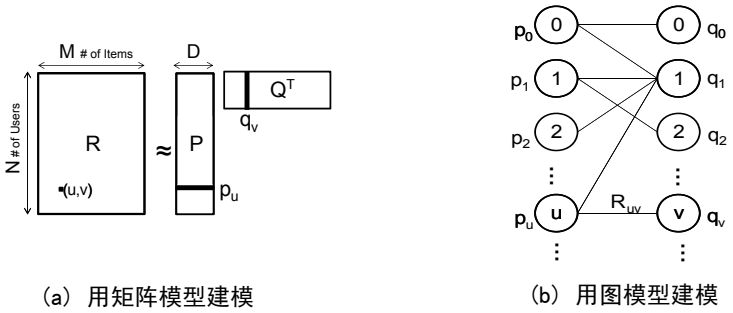
基于上述新型三维划分的技术，我们设计并实现了一个新型的分布式并行图计算框架，名为 CUBE。CUBE 由于采用了三维的任务划分因此其所需的通讯量远小于现有系统。同时，为了提升计算的性能，CUBE 还采用了前文第 2.3 节中所提到的以点为中心的编程接口和以矩阵为中心后端实现相结合的方法[15]。通过将 GraphMat 率先提出的这一方法扩展到分布式场景下，CUBE 的计算效率也比已有的分布式图计算系统要高。与此同时，由于 CUBE 对外提供的依然是以点为中心的编程接口，所以用户在使用时并没有明显的负担增加。

具体来说，本节所叙述的工作作出了如下的几方面贡献：1）我们提出了一种新型的三维图计算任务划分算法。这一三维图计算任务划分算法考虑了一个被传统一维和二维划分算法所忽略的维度。在我们的三维划分之中，数据图的点权和边权被认为是可以被进一步划分的单元，因此可能出现一个点被进一步的划分成子点并被分配给不同计算节点的情况。根据我们的实验结果，这一新型的划分方法可以极大地减少通讯量。2）我们提出了一个新型的以点为中心的编程模型 UPPS（Update, Push, Pull, Sink）。传统的以点为中心的编程模型由于并未考虑上述的第三维，它们一般假定可以以较低的代价一次性的访问同一个点的所有权值。而这一假设在三维划分的场景下是不合适的，因此我们设计了新型的编程框架与我们新型的划分算法相配合。3）我们设计并实现了一个新型的分布式图计算系统 CUBE。CUBE 采用了三维的任务划分算法并提供 UPPS 这一编程模型给用户作为其接口。同时，CUBE 会自动地将用户编写的以点为中心的程序转换成以矩阵为中心的操作进行执行，从而提升执行效率。通过我们的测试，CUBE 系统的 COST 值只有 4，远低于同类的系统[24]。4）我们系统地研究了三维划分的效果。在我们的实验中不仅仅包含微型测试用例（micro benchmark）还包括多个实际的算法应用（real-world application）。同时，我们还个给出了相应的数学公式以更加明确地计算出我们的优势。总体来说，我们的实验表明 CUBE 最高可以做到比已有同类工作中最快的 PowerLyra 系统[11] 还要快上 4.7 倍的速度（与 PowerGraph 系统[3] 相比的话可以快上最多 7.3 倍）。

## 3.2　实例研究：协同过滤问题

为了更直观地展示将点拆成子点这一被传统任务划分算法忽略的维度，这里我们考虑一个非常常见的机器学习问题：协同过滤。协同过滤问题的目标是根据已有的部分用户对商品的打分预测出如果一个用户会对一个他还未进行打分的商品进行打分的话会打几分，从而利用这个信息进行推荐。这一问题原本是使用矩阵进行定义的。具体来说，协同过滤问题的输入是一个巨大的 N 行 M 列的稀疏矩

阵 R，其中 N 为用户的数目而 M 是商品的数目。这一稀疏矩阵中的每一个非零元（i,j）表示一个已知的由用户 i 给商品 j 进行的打分。在给定输入之后，协同过滤问题的目的就在于求解出两个稠密的矩阵 P 和 Q 使它们是原稀疏矩阵 R 的一个近似分解。其中，矩阵 P 的大小为 N 行 D 列，矩阵 Q 的大小为 M 行 D 列，而近似分解的意思是 P 与 Q 的转置的乘积要尽可能的与原矩阵 R 相似（即 $R \approx P \times Q^T$）。这里 D 是特征向量的长度，也就是说在这一模型下每一个用户和每一个商品都被认为可以用一个长度为 D 的特征向量进行表示。同时一个用户 i 对一个商品 j 的打分可以用用户 i 的特征向量 P[i,:] 与商品 j 的特征向量 Q[j,:] 的点积进行预测。如果将上述的协同过滤问题转化成数据图的模型进行描述的话，很自然的每一个用户和每一个商品都会被表示成数据图中的一个点。也就是说，这张数据图会是一个二部图，原稀疏矩阵 R 中的每一个非零元 (i,j) 都会被转化成这个图中连接用户 i 的点和商品 j 的点之间的一条边，其边权就是用户 i 给商品 j 的打分。在这种情况下，很明显每一个点的点权都是一个长度为 D 的特征向量，而不是像 PageRank 那样每个点的点权都只是一个不可被进一步划分的 double。图 3.1 展示了我们上述的两种协同过滤的表示方法，可以看到，在图模型的表示下数据图中的每个点是可以被进一步的分割的。




（a）用矩阵模型建模　　（b）用图模型建模

图 3.1　协同过滤问题问题的两种表示形式。

事实上，根据我们的调研，上述协同过滤问题的这一特征并不是孤立存在的，反而是将数据挖掘和机器学习问题转化成图计算问题进行建模时的一种通用模式。举例来说，另外一个常见的例子就是稠密矩阵与稀疏矩阵的乘积（SpMM）这个在数据挖掘和机器学习领域的算法实现中非常常见的操作。在建模成图计算问题时，SpMM 问题中的稀疏矩阵会由图中的边来表示，而另外一个稠密矩阵中的每一行则代表一个点。显然这一数据图中的点也是可以被进一步划分的。原则上来说，将以矩阵为中心的应用算法建模成图计算问题时，上述可以被进一步划分的点权就会变得非常的常见。更重要的是，这些实际上是一些一维向量的点权之间进行的操作经常是所谓的对位（element-wise）操作。而对位操作的一大特点就是它们非

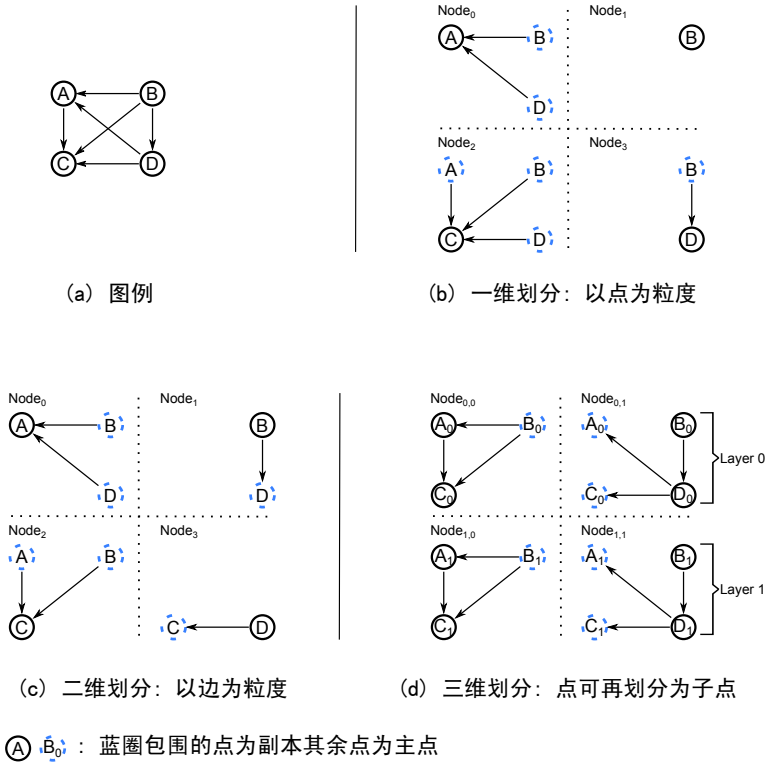



图 3.2　三种不同的划分算法示例。

常的好被并行化。只要将不同的下标的数据分配给不同的计算节点，对位操作的计算就可以以完全不引发节点间通信的方法被并行执行。这一特点进一步地表明在这个新的维度上进行划分的有效性。

## 3.3　三维划分的基本概念

作为示例，图 3.2 展示了一张小型的数据图和将其分别用一维、二维、三维的划分方法划分到一个拥有四个计算节点的集群上时最终的数据分布情况。可以看到，在一维划分的情况下，四个图中的点被均匀地分配给了四个计算节点，每个图中的点都与它所有的入边分配在同一台计算节点之上。由于这种分配方法的简单性，这种一维的划分方法被以谷歌公司的 Pregel 系统[2] 以及其开源实现 Apache Giraph[9] 为代表的一系列分布式图计算系统所采用。但是，正如我们在前文第 2.1.2 节所提到的，虽然一维的划分方法在分布比较均匀的数据图上效果尚可，但实际应用环境中所处理的真实社会中采集得到的图往往满足一种叫做 power law 的性质。在这具有这种性质的图中，点的度数分布十分的不均匀，以至于可能有一小部分的点相关的边占据了总边数的五成以上的情况。在这种情况下，如果还继续使用简单的一维划分算法的话就会在不同的计算节点之间造成极大的负载不均衡，从而影响整个系统的性能[3]。

为了解决这一问题，后续的研究人员就提出了如图 3.2 (c) 所示的二维划分方法。在二维划分方法的情况下数据图中的边被均匀地分配给不同的计算节点。由于图计算应用的复杂度一般都和边数成正比，因此在边被均匀分配的情况下负载均衡问题可以被得到较好的解决。不过，由于一个点的多条边在二维的划分方法下可以被分配个不同的节点，在这种情况下系统需要设立相关的点的副本用于通讯。而二维划分方法下数据的通讯量就与这些副本的数量成正比。为了减少副本的数量，研究人员提出了包括 Hybrid-cut[11] 在内的多种启发式的方法。然而，正如我们前文中有提到的，无论在形式上有多大的变化，这些已有的划分方法都假设数据图中的点和边都是不可划分单元，因此没有尝试对它们做进一步的划分。

不过，现有的分布式图计算框架所负责处理的问题已经不再局限于简单的图分析任务，很多数据挖掘和机器学习的任务也被建模成图计算的应用加以处理。在这种情况下，经常的数据图中的点权和边权都是一个向量，因此是可以被进一步划分的。这一新的划分维度的发现很自然地就导出了我们的新型三维划分方法。具体来说，在一个由 N 个计算节点构成的分布式集群之中，一个三维的图计算任务划分器会用到 L 份的数据图拓扑拷贝。这里 L 被要求是 N 的一个因子，即 N 除以 L 的余数要求为 0。在这种配置下，这个集群的计算节点会被均等的划分成 L 层，其中每层包含 N/L 个计算节点并负责维护数据权值向量的一部分（每个点权或者边权向量的 1/L）。同时，我们的三维图计算任务划分器会利用一个已有的二维划分方法将图的拓扑结构划分成 N/L 个子图并将其分配给一层的 N/L 个计算节点。换句话说，在三维划分的情况下，每一个点都被划分成了 L 个子点，每个包含原本点权向量的 1/L。不同子点分配到的部分不相交。对于这个由 N 个计算节点组成的集群，其第 i 层的节点负责管理由所有数据图中的点的第 i 个子点构成的数据图的一层。由于每一层的计算节点有 N/L 个，因此还需进一步的将原图划分成 N/L 个子图。这一步骤是利用一个已有的二维划分方法达成的。

通过分析可以发现，三维划分可以降低“沿着边”进行的通讯。这些通讯是由对数据图中相邻的两个点的权值向量进行对位操作造成的。由于三维划分的过程中通过增加层数 L 可以减少子图数 N/L，所以这一类的通讯量随着 L 的增大而减小。图 3.2 (d) 中给出的就是例图被三维切分后的情况，其中 N=4，L=2。在图中我们可以看到，四个不同的计算节点被分为了两层，每层 4/2=2 个计算节点。图中我们用 $\mathrm{Node}_{i,j}$ 这种标记表示这是第 i 层的第 j 个计算节点。由于关于子图的划分在不同层之间是共享的，所以对于每一层的第 j 个计算节点，即 $\mathrm{Node}_{*,j}$ 来说，它们分到的子图都是一摸一样的，只不过分到的点权向量是不相交的几段。与已有的一维和二维划分方法需要将原图拆分成四个子图不同，由于子点的大小仅为原点的 1/2，

在三维划分的情况下我们只需要将原图拆分成两个子图。这一现象也体现在了副本的数量上。由图 3.2 可见，在二维划分的情况下需要建立 6 个副本。相对的，在三维划分的情况下每层仅需建立 3 个副本。虽然因为一共有两层的原因事实上三维划分的情况下一共还是有 2*3=6 个副本。但是总的通讯量的大小实际上是正比于“每个副本的大小”与“副本数量”的乘积。在三维划分的情况下虽然总的副本的数量没有下降，但是每个副本仅是一个子点的副本，大小为原点的 1/2。所以总体上来说通讯量是减半了的。原则上来说，由于三维划分可以减少原图需要划分的子图的数量，因此层内的通信是随着第三维上划分的层数 L 的增加而减少的。但是，相对的来说，在三维划分的情况下多出了一种原本以为和二维划分都不会出现的通讯模式，即层间通讯模式。这一种层间通讯的大小是随着着第三维上划分的层数 L 的增加而增加的。一项减少、一项增加，因此在关于 L 的选取上形成了一个 tradeoff。程序员需要小心地选择最佳的 L 从而达到最佳的效果。

## 3.4　三维划分下的编程模型

已有的以点为中心的编程模型，比如说谷歌公司 Pregel 系统[2] 的点程序模型，以及为了与二维划分相配合设计的 PowerGraph 系统的 GAS 模型[3]，都是为了已有的这些传统划分算法设计的。这一历史原因导致它们都假定一个数据图中同一个点的所有权值都是放在同一个计算节点上的，因此可以很廉价地获取到，也因此他们都不适合用于三维划分的场景。所以，我们在 GAS 模型的基础上设计了一个新型的可以和三维划分算法相配合的编程模型，名为 UPPS。在这一节中，我们首先介绍 UPPS 的数据模型，即如何在 UPPS 中定义用户的图数据；然后我们会更精确的定义如何在这样的数据上进行三维的数据划分；最后我们会描述如何对定义好的数据进行操作并列出实际的应用例程。

### 3.4.1　数据模型

和其他的以点为中心的编程模型一样，UPPS 也采用数据图模型作为它的数据模型。换句话说，UPPS 的用户需要将他们的数据 D 建模成一张由点集合 V 和有向的边集合 E 组成的有向图 G。用户可以在这张有向图的各个点和边上附着任意类型的数据。到目前为止 UPPS 的数据模型都和传统的 GAS 模型的数据模型一样。

不过，不一样的地方在于：在 UPPS 中用户附着在数据图上各个点和边的数据被明确地分成了两个部分。1）一个是权值中的不可划分部分 DShare，这一部分的数据会被一个单一的数据变量存储；2）另外一部分是可划分的数据部分 DColle，而这一部分则是被存储在一个向量之中的。表 3.1 对此给出了具体的详细定义。从中

表 3.1　UPPS 编程模型。

| **Data** | |
|---|---|
| $G$ | — $\{V, E, D = \{DShare, DColle\}, S_C\}$ |
| $G_{bipartite}$ | — $\{\mathbb{U}, \mathbb{V}, E, D = \{DShare, DColle\}, S_C\}$ |
| $DShare_u$ | — a single variable |
| $DShare_{u\to v}$ | — a single variable |
| $DColle_u$ | — a vector of variable with size $S_C$ |
| $DColle_{u\to v}$ | — a vector of variable with size $S_C$ |
| $DColle_u[i]$ | — the $i^{th}$ element of $DColle_u$ |
| $DColle_{u\to v}[i]$ | — the $i^{th}$ element of $DColle_{u\to v}$ |
| $D_u[i]$ | — abbreviation of $\{DShare_u, DColle_u[i]\}$ |
| $D_{u\to v}[i]$ | — abbreviation of $\{DShare_{u\to v}, DColle_{u\to v}[i]\}$ |
| **Computation** | |
| *UpdateVertex($\mathcal{F}$)* | — **foreach** vertex $u \in V$ **do** $D_u^{new} := \mathcal{F}(D_u)$; |
| *UpdateEdge($\mathcal{F}$)* | — **foreach** edge $(u, v) \in E$ **do** $D_{u\to v}^{new} := \mathcal{F}(D_{u\to v})$; |
| *Push($\mathcal{G}$, $\mathcal{A}$, $\oplus$)* | — **foreach** vertex $v \in V$, index $i \in [0, S_C)$ **do** $DColle_v^{new}[i] := \mathcal{A}(D_v[i],\ \bigoplus_{(u,v)\in E}(\mathcal{G}(D_u[i],\ D_{u\to v}[i]))$; |
| *Pull($\mathcal{G}$, $\mathcal{A}$, $\oplus$)* | — **foreach** vertex $u \in V$, index $i \in [0, S_C)$ **do** $DColle_u^{new}[i] := \mathcal{A}(D_u[i],\ \bigoplus_{(u,v)\in E}(\mathcal{G}(D_v[i],\ D_{u\to v}[i]))$; |
| *Sink($\mathcal{H}$)* | — **foreach** edge $(u, v) \in E$, index $i \in [0, S_C)$ **do** $DColle_{u\to v}^{new}[i] := \mathcal{H}(D_u[i],\ D_v[i],\ Du\to v[i])$; |

可以看到，UPPS 的用户需要指定一个向量大小 $S_C$，即每个权值可划分部分 DColle 这个向量的大小。当只有点权的 DColle 部分被使用，边权的 DColle 部分可以被忽略时，可以设置边权的 DColle 为空（NULL）。但是，在点权和边权的 DColle 部分同时被使用的情况下，UPPS 要求每一个 DColle 向量，无论是点权的还是边权的，都保持一样的大小，即 $S_C$。这一要求之所以存在的主要原因在于我们希望进行协同地数据划分，从而达到在执行需要进行层内通讯的操作时没有任何的层间操作发生的目的。更详细的介绍可以参见后文第 3.4.3 节中对于 Push, Pull, Sink 这三个操作的描述。

### 3.4.2　UPPS 下的三维划分

通过明确地将数据图中权值的可划分部分 DColle 从不可划分部分 DShare 中独立出来，UPPS 使得任务划分可以很方便地将数据图中的每个点或边划分成几个子点或子边。其中每个子点或子边的权值包含一个完整的 DShare 的副本和 DColle 向量中的非相交的一部分。UPPS 上的三维划分可以通过首先选定一个在第三维上要划分的层数 L，然后将数据图利用一个已有的二维划分方法 $\mathcal{P}$ 划分成 N/L 个子图来实现。具体来说，UPPS 上的三维划分可以用（$\mathcal{P}$，L）表示，其中 $\mathcal{P}$ 是一个已有的二维划分算法。假定集群由 N 个计算节点构成，L 是一个能整除 N 的数，N 个节点被均匀划分为 L 层，每层包含 N/L 个节点。在此基础上，三维划分器将用算法 $\mathcal{P}$ 将边集 E 划分为 N/L 个子集，同时也将点集 V 也随机划分成 N/L 个子集。如果我们用 $\mathrm{Node}_{i,j}$ 表示第 i 层的第 j 个节点，$\mathrm{E}_j$ 和 $\mathrm{V}_j$ 分别表示 E 和 V 的第 j 个子集。划分后，$\mathrm{Node}_{i,j}$ 包含如下数据的副本：

- 一份共享的 $DShare_u$ 拷贝，如果 $u \in V_j$;
- 一份独享的 $DColle_u[k]$ 拷贝，如果 $u \in V_j$ 并且 $LowerBound(i) \leq k < LowerBound(i+1)$;
- 一份共享的 $DShare_{u\to v}$ 拷贝，如果 $(u,v) \in E_j$;
- 一份独享的 $DColle_{u\to v}[k]$ 拷贝，如果 $(u,v) \in E_j$ 并且 $LowerBound(i) \leq k < LowerBound(i+1)$;

其中，$LowerBound(i)$ 等于 $i*(\lfloor S_C/L \rfloor)+min(i,S_C\%L)$。换句话说，每一层的计算节点都拥有一份共享的 DShare 权值以及所有点和边的 DColle 权值中不相交的一份。在三维划分的过程中，L 和 $\mathcal{P}$ 共同对最终的通讯量大小造成影响。当 L 等于 N 时，计算集群的每一层都只包含一个计算节点，且这个计算节点拥有整张数据图的拓扑结构，所有的 DShare 权值，和 1/N 的 DColle 权值。在这种情况下只有一个子图，因此不需要为 DColle 权值建立任何的副本，也就是说层内通讯为零。但是，这种情况下有可能会带来很高的层间通讯，以至于综合的效率并不是最高的。相反，当 L 设定为 1 的时候，计算集群只包含拥有全部 N 个节点的一层。这个时候三维划分（$\mathcal{P}$，L）就退化成了二维划分 $\mathcal{P}$，拥有与之一样的特性。根据我们的调研，一般来说常见的最佳情况是选定一个在 2 ∼ N 的值作为 L，从而在层内通讯和层间间达成一种平衡。这种情况下 L 和 $\mathcal{P}$ 共同对最终的通讯量大小产生影响。在我们的系统中，需要程序员手工的指定这个 L 的大小。不过为了方便程序员的使用，我们提供了一系列的公式用于估算最佳的 L。详细的内容会在稍后的第 3.6 节中介绍。

### 3.4.3　计算模型

在定义了数据之后，UPPS 模型提供一共四种操作供用户操作数据。这四种操作的名字分别为 Update, Push, Pull, 和 Sink，它们的首字母组合起来就是 UPPS 这一编程模型名字的来源。表 3.1 给出了这些操作的具体定义，其中包含了所有的操作变种形式。

具体来说，Update 操作会首先收集齐需要操作的点或者边的所有权值数据，然后根据这些数据进行计算. Update 操作的输出可能是更新权值所有部分，包括 DShare 部分和所有下标的 DColle 部分。简单的说，Update 操作是一个纵向的操作，因此需要进行纵向的也就是层间的通讯。如前文第 3.4.2 节所述，在我们的三维划分方法中，一个点/边的权值会被分配给不同层的计算节点，所以对于一个 UpdateVertex 或者 UpdateEdge 操作来说，每个计算节点 $\text{Node}_{i,j}$ 都需要与其他层的对位计算节点 $\text{Node}_{,j}$ 进行通信，从而获得所需的所有数据。同样的，在进行完计算之后，每个计算节点 $\text{Node}_{i,j}$ 都还需要再一次的与其他层的对位计算节点 $\text{Node}_{,j}$ 进行通信，以发送更新后的所有数据。不过需要注意的是，在进行 Update 操作的时候，仅有层间通讯发生，没有任何的层内通讯。换句话说计算节点 $\text{Node}_{i,j}$ 不会在执行 Update 操作的过程中尝试与同一层的其他计算节点，即 $\text{Node}_{i,*}$ 通讯。

与 Update 操作相反，其余的三种操作即 Push，Pull，Sink 操作都是横向地更新数据。也就是说更新是顺着数据图中的边集合所描述的拓扑关系进行传播的。具体来说，对于一条边 (u,v)，Push 操作用源点 u 的点权和这条边的边权更新汇点 v 的点权；Pull 操作则相反，是用汇点 v 的点权和边的数据更新源点 u 的点权；相对的，Sink 操作是用两个点的数据更新边的边权。可以从表 3.1 中所描述的这些操作的具体步骤中看到，事实上 UPPS 操作集合中的 Push 操作与传统 GAS 模型中的 Gather-Apply-Scatter 操作组合基本是等价的。在 GAS 模型中，每一个数据图中的点都会使用用户指定的一个 gather 函数 $\mathcal{G}$ 对其每一条入边进行操作得到一个更新；然后会使用另一个用户指定的满足结合律和交换律的聚集函数 $\oplus$ 将这些更新综合起来得到一个唯一的更新；最后这一聚合更新被发送给这个节点的各个副本并用用户给定的更新函数 $\mathcal{A}$ 用于更新点的点权。UPPS 的 Push 操作的执行过程几乎与这是一样的，只不过在 UPPS 中，我们将每个点和边的权值中可以被划分的部分 DColle 分配给了不同层的计算节点。因此，在 UPPS 的 Push 操作中我们要求只能对对位的，也就是下标相同的 DColle 元素进行操作。换句话说，在对下标为 i 的 DColle 向量中的元素进行更新的时候用户的更新函数只能读取权值中的 DShare 部分和其它点或者边的权值中 DColle 向量部分下标同样为 i 的元素。这么做的主要目的是在执行 Push 操作的过程中仅仅进行层内的通讯而避免所有的层间

通讯。虽然对计算的能力进行了限制，但是正如我们前文中所提到的，进行对味位的操作本来就是这一类图计算应用最常见的计算模式，所以事实上我们的限制并没有影响到 UPPS 模型的编程能力。和 Update 操作相反，在进行 Push/Pull/Sink 操作的时候，仅有层内通讯发生，没有任何的层间通讯。也就是说计算节点 $\text{Node}_{i,j}$ 在执行 Push/Pull/Sink 操作的过程中只会尝试与同一层的其他计算节点，即 $\text{Node}_{i,}$ 通讯，而不会尝试与其他层的计算节点进行通讯。

### 3.4.4　二部图

上述的所有操作都是针对通用的数据图模型的。不过，根据我们的调研发现，在实际运用过程中，很多的数据挖掘和机器学习应用在建模成图计算应用的时候是用二部图作为他们的数据模型的。在二部图的情况下，数据图的点集被化分成两个不相交的子集 $\mathbb{U}$ 和 $\mathbb{V}$，并且每条边的两个端点分属 $\mathbb{U}$ 和 $\mathbb{V}$ 集合。近期有研究表明[39]，对于这一类的特殊图数据模型应该设计专用的操作以方便用户使用和提升效率。因此，我们的 UPPS 模型在原有操作的基础上另外提供了两个新的 API：UpdateVertexU 和 UpdateVertexV。这两个操作与原有的 UpdateVertex 操作所要执行的内容基本是一致的，但是分别只会更新 $\mathbb{U}$ 和 $\mathbb{V}$ 内的点，从而避免了计算的浪费。与此同时，在对二部图的数据进行三维划分的时候，我们会使用二部图专用的二维划分方法 bi-cut[39] 作为组成我们三维划分器的 $\mathcal{P}$ 的部分。

### 3.4.5　与 GAS 模型的比较

上文中也有提到，UPPS 模型在设计的过程之中也是遵从了所谓“Think as a vertex”的原则，因此在使用方面 UPPS 的编程难度与现有的以点为中心的编程模型基本保持了一致。事实上，可以认为传统图计算编程模型中的 GAS 是我们的 UPPS 模型在指定向量大小 $S_C$ 小于等于 1 的情况下的特例。因此，用户并不需要对他们已有的程序进行修改就可以直接利用我们系统的高效执行后端。相反，如果用户希望利用三维划分的优越性减少图计算应用中所需的通讯量，那么至少需要将集群分为两层。这种情况下 $S_C$ 至少要大于等二，程序的修改也是不可避免的。

然而，根据我们的调研，事实上这样的修改是非常简单的，并没有给程序员增加太多的负担。举例来说，广度优先搜索（BFS）是一个非常常用的图计算基础算法。在 GAS 模型下，BFS 的实现方法是 1）给数据图中的每一个点附上一个布尔型的点权，表示这个点是否被访问过了；2）在此基础上，每一轮 GAS 的 Scatter 阶段各个点将表示自己是否被访问了的布尔值广播给所有的后继节点；3）然后，在稍后的 Gather 和 Apply 阶段，数据图中的每个点都使用或运算对收集到的前继节

点的进行综合并与自己本身的布尔值进行操作。BFS 的过程就是不断地进行上述的操作最终达到数据图中没一个点的权值都不再变化的收敛状态。为了将这一个简单的 BFS 应用扩展成多源广度优先搜索，GAS 的用户可以简单的将 1）原来附着在数据图中每个点上的布尔变量扩展成一个由布尔变量组成的向量，其向量长度为多源广度优先搜索的源头的数量；2）在 Gather 和 Apply 的阶段使用可以对向量进行操作的对位或操作替代上述的或操作。从这里我们可以看到，在多元广度优先搜索的计算过程之中，绝大多数的计算过程都在执行一个“对位”的向量操作。经过我们的调研发现，这就是一个非常好的可以使用三维划分对程序进行优化的暗示。事实上，UPPS 的用户可以直接使用和单源广度优先搜索一摸一样的程序对多元广度优先搜索进行求解，只需将向量大小 $S_C$ 指定为源点的数量即可。同时，虽然并没有在上述的多元广度优先搜索的例子中得到展示，UPPS 模型的用户可能会需要同时访问一个点的全部点权。在数据图被进行了三维划分的情况下，为了达成用户的这一需求必然得就得引入一种新型的通讯模式，即层间通讯。而为了让用户可以对着一种新型的通讯进行表达，我们在 UPPS 模型中添加了传统 GAS 模型中没有的 Update 操作。更具体的例子将在下一节中被提到。

### 3.4.6　例程

为了更进一步的展示 UPPS 模型的使用方法，在这一节中我们提供了上文第 3.2 节中所描述的协同过滤问题的两种不同的求解算法的实现。这两种不同的算法组合起来覆盖了 UPPS 模型所有的常见使用模式。不过需要注意的是，在这一节中我们仅会在抽象的意义上描述这些操作都做了什么，具体的实现将在稍后的第 3.5 节中描述。

为了方便稍后的描述。这里我们的更明确的定义一下协同过滤问题。协同过滤问题的目的在于用已有的用户对商品的打分估测其余未被给出的打分。具体来说，如果我们使用 N 表示用户的总人数，M 表示商品的总个数，协同过滤问题的输入是一个大小为 N 行 M 列的稀疏矩阵 $R = \{R_{u,v}\}_{N\times M}$。R 中每一个非零元 R(u,v) 表示已知的一个用户 u 对商品 v 进行的打分。相对的，协同过滤问题求解后的输出则是两个稠密的矩阵 P 和 Q。P 是大小为 N 行 D 列的描述用户属性的特征数组，其中第 i 行的长度为 D 的向量就是用户 i 的特征向量。同样的，Q 是大小为 M 行 D 列的描述商品属性的特征数组，其中第 j 行的长度为 D 的向量就是商品 j 的特征向量。如果我们用 $\mathrm{Err}_{u,v}$ 来表示目前特征向量对用户 u 给商品 v 的打分的错误程度的话，$\mathrm{Err}_{u,v}$ 的计算方法就是将用户 u 的特征向量和商品 v 的特征向量进行点乘，然后再减去输入矩阵 R 中的对应项，即 $\mathrm{Err}_{u,v} = \langle P_u, Q_v^T \rangle - R_{u,v}$。总体上来说，求

解协同过滤问题的过程就是最小化累计错误值 $\sum_{(u,v)\in R} Err_{u,v}^2$ 的过程。而为了求解这一类问题，比较常用的算法有以下两种：

**GD** 梯度下降算法是所有可以用来解决协同过滤问题的算法中最经典的一个[40]。它的步骤包括首先随机初始化特征矩阵 P 和 Q 的值，然后迭代式地不断根据目前状态的梯度对 P 和 Q 的值进行优化。具体来说，特征矩阵 P 和 Q 在下一轮迭代的值为本轮迭代开始时他们的值减去 $\alpha$ 倍的当前梯度，即：

$$
\begin{aligned}
P_i^{new} &:= P_i + \alpha * (Err_{i,j} * Q_j - \alpha * P_i) \\
Q_j^{new} &:= Q_j + \alpha * (Err_{i,j} * P_i - \alpha * Q_j)
\end{aligned}
\tag{3-1}
$$

算法 1 给出了在 UPPS 模型下梯度下降算法的具体程序。其中“ + ”代表对两个输入的数进行累和。为了简单起见，在算法 1 中我们省略了用于对特征矩阵 P 和 Q 的值进行正规化的部分。在这个程序中，向量大小 $S_C$ 被设定为特征向量的长度 D，因此数据图中每一个点和每一条边的权值都包含一个长度为 D 的 *double* 数组/向量。对于数据图中的这些点，这一元素为 *double* 类型的数组存储的就是这个点对应的用户或者商品的特征向量。相对的，对于数据图中的这些边，每条边的权值中的向量都会被用于存储计算特征向量点乘过程中的中间结果。另外，在数据图中不可被进一步划分的共享变量部分，每一条边（u，v）边权的 DShare 都是一个 {*double* rate, *double* err} 的 pair。其中 rate 部分存储的是用户 u 对商品 v 的打分，err 则是存储的在这一轮迭代的情况下对这个打分的预测错误幅度。相反的，数据图中的点权仅包含 DColle 部分，而没有 DShare 部分。在数据被定义为如上的形式之后，算法 1 的计算过程基本上就是上述更新公式的一个直接翻译。在第一部中，算法 1 通过如下的步骤计算每一个条边的 err 权值：1）使用一个 Sink 操作对每一条边连接的两个特征向量计算对位的元素的乘积，并将这一中间结果存储在这条边的 DColle 数组中；2）使用一个 UpdateEdge 函数将每一条边的 DColle 数组中存储的中间结果进行累和，并减去这条边的 rate 值从而得到 err 并存储在对应的 DShare 部分中。在计算完每一条边的 err 值之后，上述公式 3-1 中所描述的更新公式就可以简单的直接用一个 Pull 和一个 Push 操作完成了。正如算法 1 中所展示的，UPPS 的用户在实现 Push，Pull，Sink 这三个函数所需的用户定义函数时仅需操作输入的各个点和边的权值中的 DColle 部分的某一个对位的元素。相对的，UpdateVertex 和 UpdateEdge 函数的用户定义函数可以操作整个 DColle 向量。这与我们在前文第 3.4.3 节中的叙述是一致的。更重要的是，可以发现，用户事实上并不需要关心这些 DColle 的元素是如何分布的，比如说是否是分布在同一台计算节

点之上的。相关的通讯操作会由我们的系统自动地完成，从而方便程序员的使用。

**算法 1** 梯度下降算法在 UPPS 模型下的实现。

**Data:**

$S_C :— D$

$DShare_u :—$ NULL;　$DShare_{u\to v} :— \{double\ Rate, double\ Err\}$

$DColle_u, DColle_{u\to v} :— vector<double>(S_C)$

**Functions:**

$F_1(u_i, v_i, e_i) :— \{return\ u_i.DColle[i] * v_i.DColle[i];\}$

$F_2(e) :— \{$

$e.DShare.Err := sum(e.DColle) - e.DShare.Rate;$

$return\ e;$

$\}$

$F_3(u_i, e_i) :— \{return\ e_i.DShare.Err * u_i.DColle[i];\ \}$

$F_4(v_i, \Sigma) :— \{return\ v_i.DColle[i] + \alpha * (\Sigma - \alpha * v_i.DColle[i]);\}$

**Computation for each iteration:**

$Sink(F_1);$

$UpdateEdge(F_2);$

$Pull(F_3, F_4, +);$

$Push(F_3, F_4, +);$

ALS 除梯度下降算法之外，Alternating Least Squares (ALS) 也是一个非常常用的求解协同过滤问题的算法[41]。与梯度下降算法不同，ALS 算法初始时仅需随机初始化两个特征矩阵 P 和 Q 中的一个。随后，ALS 会在迭代的过程中碾转的通过一个特征矩阵的值求解出当前最优的另外一个特征矩阵的值。这一方法通过在每一轮迭代的过程中都固定两个特征矩阵中的一个的值，将原本非常难以求解的非图优化问题转化成了一个求解多元二次目标函数最优解的问题。而后者在数学上拥有复杂度较低的确定性解法。具体来说，ALS 的过程分为如下的几个步骤：

**Step 1** 随机初始化特征矩阵 P 的值

**Step 2** 固定 P 的取值，求解出在这一前提条件下最优化的特征矩阵 Q 的值。这一步骤可以通过将 Q 的值设置为 $Q_v = (\sum_{(u,v)\in R} P_u^T P_u)^{-1}(\sum_{(u,v)\in R} R_{u,v} P_u^T)$ 完成。

**Step 3** 在步骤二求解出 Q 的值之后，第三步和第二步在过程上基本上是一样的。只不过第三步中是固定 Q 的值，并用类似的方法求解出在这一前提条件下最优化的特征矩阵 P 的值。

**Step 4** 交替迭代执行步骤二和步骤三，直到收敛。

作为一个典型的建模成二部图的机器学习算法用例，在实现 ALS 算法的过程

中我们使用了前文第 3.4.4 节中所提到的针对二部图设计的专用 API。在 UPPS 下的具体算法实现由算法 2 给出，其中同样省略了将特征矩阵正规化的部分。从程序中我们可以看到，与梯度下降算法中将向量大小 $S_C$ 设置成特征向量长度 D 不同，ALS 算法的实现将 $S_C$ 设置成了 D+D*D。在这种情况下，数据图中的每一个点权的 DColle 事实上包含两个部分：1）一个依然长度为 D 的数组 Vec，用来存储对应用户或者商品的特征向量；以及 2）一个大小为 D*D 的矩阵缓存 Mat，用来存储 $Vec^t * Vec$。

**算法 2** ALS 算法在 UPPS 模型下的实现。

```
Data:
    S_C :— D + D * D
    DShare_u :— NULL;    DShare_{u→v} :— {double Rate}
    DColle_u :— vector<double>(S_C);    DColle_{u→v} :— NULL
Functions:
    F_1(v) :— {
        foreach (i, j) from (0,0) to (D − 1, D − 1) do
            v.DColle[D + i * D + j] := v.DColle[i] * v.DColle[j];
        return v;
    }
    F_2(u_i, e_i) :— {
        if i < D do return e_i.DShare.Rate * u_i.DColle[i];
        else return u_i.DColle[i];
    }
    F_3(v) :— { DSYSV(D, &v.DColle[0], &v.DColle[D]); return v;}
Computation for each iteration:
    UpdateVertexU(F_1);
    Push(F_2, +, +);
    UpdateVertexV(F_3);
    UpdateVertexV(F_1);
    Pull(F_2, +, +);
    UpdateVertexU(F_3);
```

在以上述的方式对 ALS 的算法进行了定义之后，ALS 算法过程中的每一个步骤二都可以按照图 3.3 所示的方法实现。从图中我们可以看到，在初始的情况下只有特征矩阵 R 说转化成的点集 $\mathbb{U}$ 的点权中的 Vec 部分的值是有效的，因此其他部分的权值都用? 表示。然后，用户可以调用一个 UpdateVertexU 函数去计算每一个点的点权对应的 $Vec^t * Vec$ 并将结果存在对应点权的 Mat 部分。紧接着，一

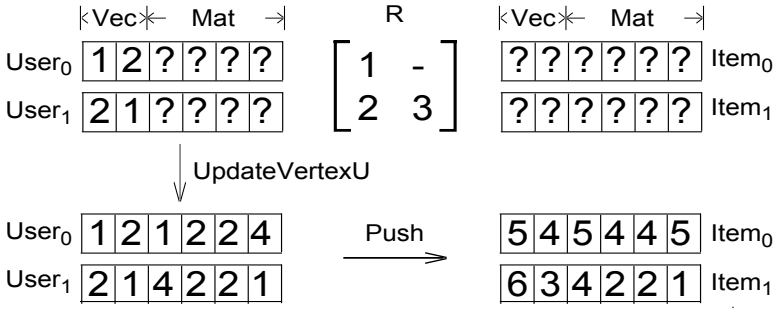



图 3.3　ALS 算法第二步的具体实现方法。

个 Push 操作被用于计算所需的累和。对于点集 $\mathbb{V}$ 中的每一个点 v，$\sum_{(u,v)\in R} R_{u,v} P_u^T$ 和 $\sum_{(u,v)\in R} P_u^T P_u$ 被分别的计算并存储在点 v 点权的 Vec 和 Mat 部分。最后，用户通过调用一个 UpdateVertexV 操作计算上文中描述的最终结果。具体的方法是调用 LAPACK 库[42] 中的 DSYSV 函数。与图 3.3 描述的 ALS 算法的第二步类似，ALS 算法的第三步可以用一系列的 UpdateVertexV，Pull，和 UpdateVertexU 函数实现。

## 3.5　系统实现

为实现 UPPS 模型，我们设计并基于 C++ 和 MPICH2 实现了一个新型的图计算引擎，其名为 CUBE。在本节中，我们首先描述 CUBE 系统中数据的载入和划分过程。然后我们将分别介绍 UPPS 模型中定义的各种操作的实现。最后，我们将简述我们是如何利用矩阵对系统的 cache 利用率进行优化的。

### 3.5.1　数据载入和划分

CUBE 采用的是完全对称的系统架构，并不存在一个中心的主节点。因此，在系统初始化阶段，运行 CUBE 的集群中的每一个计算节点都会独自加载图数据中不相交的一部分。在数据并行载入之后，CUBE 就会按照前文第 3.4.2 节中描述的对数据进行三维划分并分配给相应的计算节点。

具体来说，CUBE 的用户需要指定一个第三维上划分的层数 L 和一个二维划分方法 $\mathcal{P}$。一般来说 L 的最佳值取决于具体的图数据的性质，因此无法事先得知。相对的，我们默认采用由上海交通大学的研究人员提出的 hybrid-cut 算法作为针对一般图的划分方法[11]。同时，在图数据为二部图的情况下，另外一个 bi-cut 划分算法[39] 则会被默认启用。之所以选用这两个算法的原因在于他们都有很简单的分布式实现，并且在实际的应用例子中取得了很好的效果。在确定 $\mathcal{P}$ 和 L 之后，CUBE 会利用 $\mathcal{P}$ 将数据图划分成 N/L 个不相交的边的集合，分别为 $E_0 \sim E_{N/L}$。同时 CUBE 还会将数据图中所有的点划分成 N/L 个不相交的点的集合 $V_0 \sim V_{N/L}$。在得到了这样的一个划分之后，CUBE 会进行一一次全局的 shuffle 操作将数据分配给应该获得

的计算节点。具体来说，在 shuffler 阶段之后，每个计算节点 $\text{Node}_{i,j}$ 将存储有 $D_a[k]$ 和 $D_{b\to c}[k]$，其中 $a \in V_j$，$(b \to c) \in E_j$，并且 $LowerBound(i) \le k < LowerBound(i+1)$。

### 3.5.2　Update 操作的实现

在 Update 操作的执行过过程中，对于数据图中每一个点或者每一条边的操作都需要能够读取和修改其全部的权值。因此，CUBE 会为数据图中的每一个点和每一条边指定一个计算节点作为其 master。在 Update 操作的执行过程中，每个点或者每一条边的 master 会首先 1）负责从其它层的计算节点中收集这个点/边所有的权值；2）将用户定义的 update 函数 $\mathcal{F}$ 作用于收集到的完整权值之上；最后 3）将更新后的权值重新发给这些不同层的计算节点。对于一个一般的图，每一次 Update 操作会更新数据图中所有的点。相对的，针对二部图的 Update 操作 UpdateVertexU 和 UpdateVertexV 只会更新对应的点集 $\mathbb{U}$ 和点集 $\mathbb{V}$ 中的点。

正如之前所定义的边集 $E_j$ 和点集 $V_j$ 是分别由指定的二维划分方法 $\mathcal{P}$ 和随机划分方法划分出来的第 j 个边的集合和第 j 个点的集合。这一划分会被不同层的对位的计算节点即 $\text{Node}_{*,j}$ 所共用。在执行 Update 操作的过程中，这一点集和边集会被进一步的划分成 L 个子集合并分配给不同层但是对位的计算节点，用于指定这些计算节点是哪一些点和边的 master。具体来说，CUBE 会使用 $\mathcal{P}$ 将每一个边的集合 $E_j$ 进一步的划分成 L 个子集合 $E_{0,j}\sim E_{L-1,j}$ 并将他们分别分配给 $\text{Node}_{0,j}\sim\text{Node}_{L-1,j}$。同理，点的集合 Vj 会被再一次的随机划分成 L 个子集合 $V_{0,j}\sim V_{L-1,j}$ 并分配个相应计算节点。这样的划分保证了在执行 Update 操作的过程中仅会执行层间的通讯。每次两轮，分别为一次 gather 用于收集所有的权值，和一次 scatter 用于广播更新后的权值。

除了上述一般情况之外，在某些特定的情况下可能用户并不需要得到所有的权值，而是只需要权值的某种综合表达形式。比如说，在实现梯度下降算法的过程中，其 UpdateVertex 操作所用的用户自定义函数事实上只需要知道每条边的 DColle 向量的累和就可以了，并不需要知道每一个元素确切的值。在这种情况下我们允许用户定义一个 local combiner 用于综合本地的 DColle 权值，从而进一步的降低通讯量。在提供了 local combiner 的情况下，每一层的节点仅仅需要将综合后的结果发送给对应的点或者边的 master，而不用发送所有的权值。更进一步的，对于特殊的一些情况，我们还允许用户直接设定对应的 MPI_OP，并利用 MPI 提供的 MPI_Allreduce 原语去进一步地降低通讯量。

### 3.5.3　Push，Pull，和 Sink 操作的实现

为了保证 Push，Pull 和 Sink 操作的执行，CUBE 系统会在需要的时候为数据图中的点设立副本。具体来说对于每一个计算节点 $\text{Node}_{i,j}$ 对其所分配的边集 $E_j$ 所包含的每一条边（u,v）其中包含的两个点 u 和 v 要么同时被这个计算节点说分配到的点集 $V_j$ 所包含，要么 $\text{Node}_{i,j}$ 上有一个这个点的副本。在执行 Push，Pull 和 Sink 操作之前，一轮 L 次并行的层内 AlltoAll 通讯会被用于将这些副本同步到最新的状态，以便后续的执行。在同步之后，对于 Pull 和 Push 操作，用户给定的自定义 Gather 函数 $\mathcal{G}$ 将被用于每一条边以得到这条边在这一轮产生的更新。其后，还需要再进行 L 个 $\text{Node}_{i,*}$ 之间的 AllToAll 通信对计算结果进行归约，归约函数为 $\oplus$。为了节省通讯量，这一规约函数也会被用于本地的规约。换句话说，如果一个点对应的多条入边被分配给了同一个计算节点，这些边对应的更新会首先在本地被归约，然后只有规约后的结果才会被传输。对于一般的图来说，所有的节点的副本都会在 Push 和 Pull 的准备阶段被同步。相反，对于二部图，只有 $\mathbb{U}$ 和 $\mathbb{V}$ 中的某一个点集的副本会被同步。此外，Sink 操作的执行相对来说比较简单。在同步完成之后将用户自定义的 $\mathcal{S}$ 函数做用于每一条边即可。

### 3.5.4　基于矩阵的数据结构

前文第 2.3 节有提到，以点为中心的计算模型相对来说更加易用，但其性能比基于矩阵的执行引擎大概要差 2 倍到 6 倍左右[15,32,43]。因此，为了同时拥有以点为中心的编程模型的易用性和以矩阵为执行后端的高性能，CUBE 会自动地将以点为中心的程序转化成矩阵操作执行。具体来说，在 CUBE 中数据图的点权和边权都是被存储在连续的数组结构中的。边权的数据被等价成稀疏矩阵的数据并用 COO 的格式进行存储，即每一条边都被表示为一个包含（源点编号，汇点编号，边权）的三元组，并且所有的边是被连续的存储在一个数组中的。同理，点权的数据也是直接按照它们点编号的大小连续存储的。不过，这里就涉及到一个对于点的点权进行寻址的问题。由于每个计算节点仅仅被分配到了一部分的图，所以很有可能其被分配到的点的集合的编号是不连续的。比如说在图 3.4 给出的例子中，对于被分配到第一个子图 sub-graph 1 的节点来说点 C 是不存在的。在这种情况下设计一个高效的寻址方法就是很必要的。传统的图计算引擎中经常使用哈希表的方法对寻址信息进行存储。虽然简便，但是这种方法的效率不高。相反的，CUBE 会实现将不连续的全局点编号转换成连续的局部点编号，因此在寻址是可以直接通过下标访问。由于 CUBE 会事先纪录相应的映射关系，并将已有的边中的点的编号信息都提前转换，上述的转换工作仅需在 CUBE 系统初始化的阶段进行，没有后续

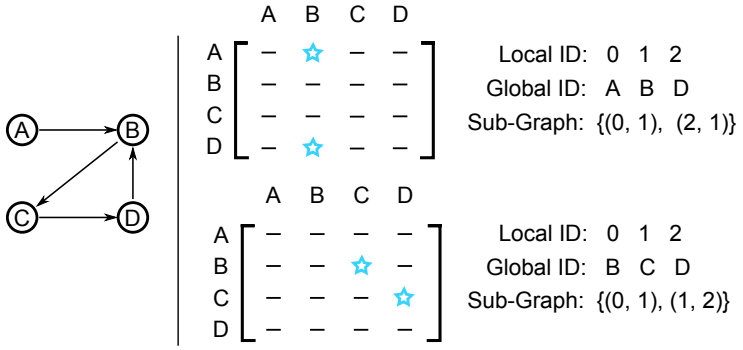



图 3.4　CUBE 系统的矩阵后端。

的开销。具体的转换方法可以参见图 3.4。除上述方法之外，CUBE 还利用 Hilbert order[44] 对点数据的访问局部性进行了进一步的优化。正如前文第 2.3 节中所提到的，与常用的只能保证对源点数据的访问局部性的字典序不同，按照 Hilbert order 进行排序的边集合可以同时优化对源点数据和对汇点数据的访问局部性，因此可以获得更好的性能。

## 3.6　实验结果

本节中我们将详细地描述 CUBE 系统的测试结果并将其与本领域内目前最好的系统 PowerLyra 和 PowerGraph 进行对比。于我们测试的每一个例子，我们都会同时给出 1）可以用于估测最终通讯量结果的数学公式；以及 2）验证了我们给出的数学公式的准确性的实际测试结果。为了能够提供更加全面的测试结果，除了通讯量和总体运行时间之外，我们还给出了有关可扩展性、内存消耗、划分算法的开销、和 COST 系数等指标的测试结果。

### 3.6.1　测试环境

我们的实验环境是一个由 8 个计算节点（Intel® Xeon® CPU E5-2640）组成的系统，其中每个节点都包含 8 个主频为 2.50Ghz 的核，节点之间用带宽 1Gb 的以太网连接。为了尽可能的模拟实际的应用环境，我们的测试数据集使用的是由斯坦福大学 SNAP 项目[45] 所收集的真实数据集。具体包括 Libimseti，LastFM 和 Netflix 这三个。表 3.2 给出了关于这三个数据集特征的具体描述。

前文第 3.4.2 节中有提到，我们的三维划分算法依赖于一个二维划分算法构成。因此，我们首先在选取的这些数据集上测试了目前 PowerGraph 和 PowerLyra 这两个系统所提供的所有二维划分算法的划分结果。其中就包含基于启发式规则的一般图划分算法 hybrid-cut ，主要针对二部图的划分算法 bi-cut，和一系列基于哈希的随机算法。对于每一种算法，我们都计算了其产生的划分结果中平均每一

表 3.2　实际图数据集合。

| 数据集 | $\|\mathbb{U}\|$ | $\|\mathbb{V}\|$ | $\|E\|$ | 最好的二维划分算法 |
|---|---|---|---|---|
| Libimseti[46] | 135,359 | 168,791 | 17,359,346 | Hybrid-cut |
| Last.fm[47] | 359,349 | 211,067 | 17,559,530 | Bi-cut |
| Netflix[41] | 17,770 | 480,189 | 100,480,507 | Bi-cut |

个数据图中的点具有的副本数量 $\lambda$。为方便起见，我们的 $\lambda$ 是包括原本的点的，所以在只有一张子图，即不存在附加的副本的情况下，其取得最小值为 1。通过比较在同一张图上不同划分算法产生的 $\lambda$ 的大小我们选取其中最小的那一个作为对于这一张图来说最佳的二维划分算法。具体的结果同样在表 3.2 中有展示。在下文中，我们会使用 $\lambda_x$ 表示在划分成 x 各自图的情况下的平均副本数量（$\lambda_1 = 1$）。从表 3.2 中我们可以看到，对于 Libimseti 这个数据集，hybrid-cut 是效果最好的二维划分方法。相对的，对于其他两个数据集 LastFM 和 Netflix，其最好的划分方法都是 bi-cut。只不过对于使用 bi-cut 时所需要指定的 favorite set 这一参数，对于 LastFM 需要指定为源点集合 $\mathbb{U}$，而对于 Netflix 数据集需要指定为汇点集合 $\mathbb{V}$。

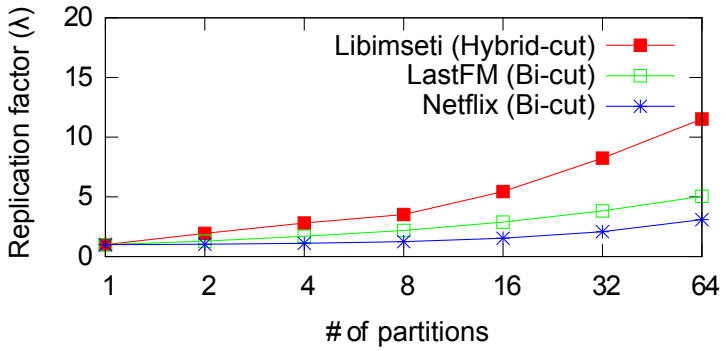



图 3.5　各图数据集在使用最佳的二维划分方法时产生的 $\lambda$ 的大小。

图 3.5 展示了这三个数据集合各自在使用其对应的最佳的二维划分方法的情况下被划分为 x 个子图时的 $\lambda$ 值。从中我们可以看到，bi-cut 算法的划分结果的好坏取决于这个二部图两个点集大小的不平衡程度。对于 Netflix 数据集，其汇点集合的大小大约为原点集合的 27 倍。这种情况下 bi-cut 方法的效果非常良好，以至于 $\lambda$ 随着子图数量 x 的增长曲线幅度比较平缓。即使是在 x = 64 即将原图划分成了 64 个子图的情况下，平均下来每个点包含的副本数量 $\lambda_{64}$ 也不过 3.04。相对的，由于 LastFM 数据集这两个点集的大小较之 Netflix 数据集更加均衡，在这个数据集上 bi-cut 的表现就要差上一些。具体表现可以参见图 3.5 中的曲线，表示 LastFM 的 $\lambda$ 增长曲线的增长幅度较之 Netflix 更加陡峭。同理，在 Libimseti 这个原点集合大小基本上和汇点集合大小相等的数据集上，专门为二部图所设计的 bi-cut 算法

的表现就连为一般图设计的 hybrid-cut 都比不上了。其对应的 $\lambda$ 增长曲线也最为陡峭。根据我们的调研发现，我们新提出的三维划分算法的效果与其所给予的二维划分算法的效果成负相关。换句话说，在目前已有的二维划分算法已经能够取得很好效果的 Netflix 数据集上我们的三维划分算法能够达到的加速比最低。相对的，对于已有的二维划分算法不能达成很好的结果的 Libimseti 数据集，我们的三维划分方法可以取得最好的加速比。在随后的章节之中我们会详细地对这一现象的产生原因进行论述。

### 3.6.2　微型测试集

CUBE 系统最大的优势就在于允许用户通过调整系统在第三维上的划分层数 L 对层内通讯和层间通讯的大小进行调节，从而达到最少的总通讯量。举例来说，两种极端的 L 设置方法是：1）将 L 设置成 1。因为只有一层，此时只有层内通讯而没有层间通讯。三维划分的实际效果也退化成和只用二维划分时一样；2）另一种极端情况则是将 L 设定为和集群中的计算节点的数量 N 一样。在这种情况下每一层只有一个计算节点因此层内通讯量为零，只有层间通讯。总体上来说，随着 L 的值的不断增大，层内通讯的数量逐渐减小而层间通讯的数量逐渐增大。用户需要选取一个合适的 L 以达到这两者间的平衡。为了更加清晰地展示这一变化过程，在本节中，我们给出了关于三个微型测试用例（micro benchmark）的测试结果。其中包含具体的可以用来计算总通讯量数学公式和在 $N = 64$ 的情况下将 L 从 1 变化到 64 的各种情况下的最终运行时间。之所以在使用实际算法进行测试之前首先使用这些微型测试用例的原因有两个。首先，这几个微型测试用例分别都只需要使用我们的 UPPS 模型给出的所有操作中的一个进行计算，因此可以非常明确地展示 L 的取值对这一操作的执行时间的影响。其次，我们给出的用于计算这些微型测试用例通讯量的数学公式可以被用来组合成计算复杂应用的通讯量的数学公式。而我们将在本节中展示这些攻势对实际结果的精准预测能力。

#### 3.6.2.1　SpMM

稀疏矩阵与稠密矩阵的乘积这一操作的具体定义为将一个稠密但相对较小的矩阵 A （大小为 $D \times H$）与一个较大但稀疏的矩阵 B（大小为 $H \times W$）相乘并计算其结果。一般来说 D 的取值要远远小于 H 和 W 的取值。这一计算核心非常普遍的被应用于各种数据挖掘和机器学习的算法之中。举例来说，在深度学习算法的训练阶段，其神经网络的参数矩阵就可以看作一个大的稀疏矩阵，而以 min-batch 形式输入的训练数据就可以看做是一个较小的稠密矩阵[48]。其中 D 为这个 mini-bach

的大小，一般取值为 20 到 100。在这种情况下，神经网络的训练阶段主要做的事情就是进行稀疏矩阵和稠密矩阵乘积的计算。在利用 UPPS 模型对这一应用进行建模的时候，数据图可以是一张拥有 H + W 个点的二部图，其中 $|\mathbb{U}| = H$，$|\mathbb{V}| = W$。稀疏矩阵中的每一个非零元（i,j）都被表示成这张二部图中由点集 $\mathbb{U}$ 中的点 i 指向点集 $\mathbb{V}$ 中的点 j 的一条边。这条边的边权中 DShare 的部分为这个非零元的值 $b_{i,j}$，而 DColle 部分不被使用，即 DColle = NULL。在另一方面，稠密矩阵 A 由这个数据图中的各个点表示，其中 A 的第 i 列被点集合 $\mathbb{U}$ 中的第 i 个点表示，其点权的 DColle 部分为一个长度为 D 的向量（即 $S_C = D$）而点权的 DShare 部分为 NULL 不被使用。在进行了这样的数据建模之后，整个稀疏矩阵与稠密矩阵乘积的操作就可以用一个 Push 操作来实现。

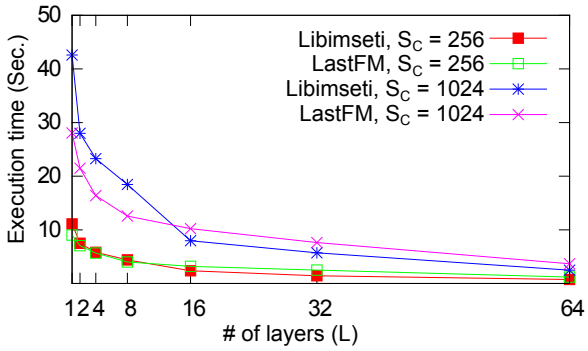



图 3.6　SpMM 在 L 取不同的值时的执行时间。

图 3.6 中给出了我们在一个由 64 个计算节点的集权上将 L 从 1 变化到 64 的每种情况下 SpMM 操作的执行时间。由于 SpMM 操作的计算非常的简单，图上所可以看到的运行时间的缩减完全是由通讯量的减少所带来的。具体来说，如果一个三维划分算法（$\mathcal{P}$, L）被用来讲一张图划分给一个拥有 N 个计算节点的集群，其最终每一层中会拥有的副本的数量为 $\lambda_{N/L} * |V|$。由于在执行 Push 和 Pull 操作的过程中仅仅进行层内通讯，其目的也仅限于将层内的所有点权的 DColle 数据的副本与其对应的 master 进行同步，其中共需要的通讯量可以用 $(S_C/L) * (\lambda_{N/L} - 1) * |V|$ 这个公式计算得到。

在二部图的情况下，为了进一步的优化通讯量，我们每次仅仅同步两个点的集合 $\mathbb{U}$ 和 $\mathbb{V}$ 中一个点集的副本。因此，如果 SpMM 操作是由 Push 操作实现的，其具体的通讯量大小为 $(S_C/L) * (\lambda^{\mathbb{V}}_{N/L} - 1) * |\mathbb{V}|$。相反，如果 SpMM 操作是由 Push 操作实现的，其具体的通讯量大小为 $(S_C/L) * (\lambda^{\mathbb{U}}_{N/L} - 1) * |\mathbb{U}|$。在这两个公式中，$\lambda^{\mathbb{U}}_{N/L}$ 和 $\lambda^{\mathbb{V}}_{N/L}$ 分别代表点集 $\mathbb{U}$ 中所有点的平均副本数量和点集 $\mathbb{V}$ 中所有点的平均副本数量。由上述推导我们可以最终得出，对于一个 SpMM 操作，其总共所需的通讯量可以由以下的公式计算得出：

$$\text{Traffic}(\text{SpMM}_{\text{Push}}) = 2 * \text{S} * S_C * (\lambda_{N/L}^{\mathbb{V}} - 1) * |\mathbb{V}| \tag{3-2}$$

$$\text{Traffic}(\text{SpMM}_{\text{Pull}}) = 2 * \text{S} * S_C * (\lambda_{N/L}^{\mathbb{U}} - 1) * |\mathbb{U}| \tag{3-3}$$

其中 S 表示每一个 DColle 向量中所包含的元素 DColle[i] 的大小。通讯量比之前提到的公式乘以了一个二的原因在于在 Push/Pull 操作中每一次的执行包含两轮的 AllToAll 通讯。与上述二部图的特殊情况不同，如果针对的是一张普通的图，Push 和 Pull 操作的通讯量是相等的，均为：

$$\text{Traffic}(\text{Push}/\text{Pull}) = 2 * \text{S} * S_C * (\lambda_{N/L} - 1) * |V| \tag{3-4}$$

在使用 hybrid-cut 作为 $\mathcal{P}$ 的情况下，将 Libimseti 划分成两个子图时 $\lambda_2$ 等于 1.93。相对的，将 Libimseti 划分成 64 个子图时 $\lambda_{64}$ 等于 11.52。因此，由如上的公式可以推算出通过将 L 设置成 32 时可以减少 $1 - (0.93/10.52) = 91\%$ 的通讯量。从图 3.6 中我们也可以看到，在 $S_C$ 等于 256 的情况下把 L 从 2 提升到 32 的过程中整体的运行时间降低了 7.78 倍。如果把 $S_C$ 提高到 1024 的时候加速比的具体数值则变为 7.45 倍。

#### 3.6.2.2　SumV

在 SpMM 这个例子下，最佳的运行时间永远是在将 L 设定为 N 的情况下取得的。因为 SpMM 操作可以由一个 Push 或者一个 Pull 操作实现，其中所用的通讯均为层内通讯，没有层间通讯。相对的，如果一个应用只有层间通讯没有层内通讯，那么其结果就是运行时间睡着 L 的增大而不断增大。为了展示这一情况，本节给出了另外一个微型测试用例 SumV 的测试结果。SumV 执行的操作本身只是将数据图中每一个点的点权中 DColle 向量的所有元素累和并存储到这个点权的 DShare 部分（即 $DShare_u := sum(DColle_u)$）。显而易见，这一操作可以用一个 UpdateVertex 操作实现。同时，因为这一操作满足前文第 3.5.2 节中所提到的可以进行局部本地综合的优化，因此理论上可以通过增加一个 loca combiner 对通讯量做进一步的优化。不过本节只要想展示一般情况下层间通讯随 L 的变化趋势，所以在我们的测试用例中并没有实现这个 local combiner。

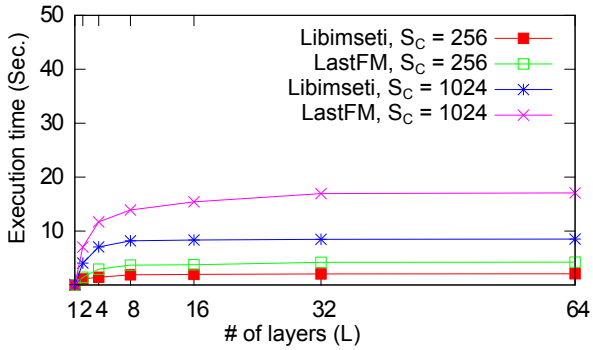



图 3.7　SumV 在 L 取不同的值时的执行时间。

图 3.7 展示了在一个 64 个计算节点的集群上执行 SumV 操作并将 L 从 1 逐步增加到 64 的情况下各组实验的测试结果。从中可以看到随着 L 的增加 SumV 的总体运行时间不断增大，这与我们之前的推想是吻合的。同时我们还可以看到，随着 L 的增加这条曲线的斜率是在不断降低的。为了解释这一现象，我们同样给出了可以用于计算 SumV 操作所需通讯量的公式。具体来说，为了执行 UpdateVertex 操作，每一个计算节点 $\text{Node}_{i,j}$ 都需要收集他所管理的所有点的 master，即 $V_{i,j}$ 中的点的点权的所有 DColle 元素。由于 $V_{i,j} \subseteq V_j$，总共需要收集的数据大小可以由 $S_C * |V_{i,j}| - \frac{S_C}{L} * |V_{i,j}| = \frac{L-1}{L} * S_C * |V_{i,j}|$ 这个公式所计算得到。在收集并计算完成之后，$\text{Node}_{i,j}$ 还需要将这些所有的结果发送给其他层的节点（包括 DColle 和 DShare）。所以，总体上来说每一次执行 UpdateVertex 操作所需的通讯量可以由如下的这个公式计算得到：

$$\begin{aligned}\text{Traffic(SumV)} &= \text{Traffic(UpdateVertex)} \\ &= 2 * \text{S} * \frac{L-1}{L} * S_C * |V| + \text{S} * (L-1) * |V|\end{aligned} \tag{3-5}$$

我们可以从上述的这个公式中看到，在 $S_C$ 足够大的情况下，上述公式的最终结果由公式的前半部分所主导。在这种情况下，显然随着 L 的增加这条曲线的斜率是在不断降低的。由于 SumV 操作的计算比较简单，其总体运行时间基本上被网络通讯时间所主导。因此我们可以在图 3.7 中看到一条与上述公式所近似的曲线。

#### 3.6.2.3　SumE

与 SumV 类似，SumE 也是一个简单的将权值中的 DColle 部分进行累和并将结果存储到 DShare 部分的应用。只不过在 SumE 中被累和的所有边的权值，因此 SumE 是由一个 UpdateEdge 实现的。图 3.8 给出了这一应用总体执行时间随着 L

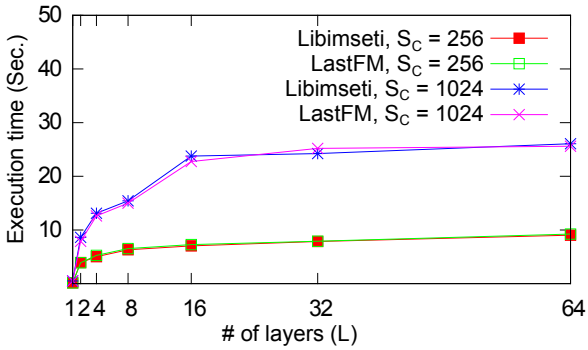



图 3.8　SumE 在 L 取不同的值时的执行时间。

的变化趋势。可以看到，其曲线基本上和 SumV 的相同。同样的我们可以通过推到得到 SumE 的通讯总量为：

$$\begin{aligned}\text{Traffic(SumE)} &= \text{Traffic(UpdateEdge)} \\ &= 2 * \text{S} * \frac{L-1}{L} * S_C * |E| + \text{S} * (L-1) * |E|\end{aligned} \tag{3-6}$$

#### 3.6.2.4　小结

从上述三个微型测试用例的结果中我们可以看到，总体上来说，随着 L 的值的不断增大，层内通讯的数量逐渐减小，因此 Push/Pull/Sink 这三个操作的执行时间不断减少。相对的，层间通讯的数量会逐渐增大，所以 UpdateVertex 和 UpdateEdge 函数的执行时间会不断变大。因此，在会同使用到这两种操作的实际应用场景下，用户需要选取一个合适的 L 以达到这两者间的平衡。

### 3.6.3　实际应用

除了上一节中提到的三个微型测试用例之外，我们还用 CUBE 实现了梯度下降和 ALS 这两个实际的应用例子作为 CUBE 的测试用例。其中，梯度下降算法同时使用了包含层内通讯的 Sink 操作和包含层间通讯的 UpdateEdge 操作。而 ALS 算法则是使用了剩下的包含层内通讯的 Push 和 Pull 操作，以及包含层间通讯的 UpdateVertex 操作。梯度下降算法的 UpdateEdge 操作可以用 local cominer 进行优化，而 ALS 则是使用了针对二部图优化的两个特殊 UpdateVertex 操作。所以，总的来说，这两个算法包含了 UPPS 模型的所有基本使用模式，以至于其它更加复杂的算法也可以被看成是这两个算法的某种组合。举例来说，神经网络的学习算法就可以被看作是一轮用来求解 loss function 的 ALS 轮和一轮用来更新参数的梯度下降轮。在本节余下的内容中，我们将首先展示我们的视线相对于已有的系统

PowerGraph 和 PowerLyra 来说 CUBE 在性能上能有多大的优势。随后，我们将详细讨论则以加速比的主要组成部分，以及具体我们节省了多少的通讯量。

#### 3.6.3.1　实现

对于梯度下降和 ALS 这两个算法，PowerGraph 系统和 PowerLyra 系统都已经提供了他们各自的实现。所以在进行对比测试的时候我们可以直接使用这些代码。关于划分算法的选取，PowerGraph 系统我们使用的是默认的 oblivious[3] 划分算法。而对于 PowerLyra 系统，我们使用的是如表 3.2 所示的经过枚举测试发现的最佳的二维划分算法。对于 CUBE 系统，我们对梯度下降和 ALS 算法的实现基本上和前文第 3.4.6 节中描述的一样，只不过我们进行了一些进一步的优化。对于梯度下降算法，我们添加了一个 local combiner 使得系统可以将 DColle 的数据先在本地累和后再进行传输。而对于 ALS 算法，我们分别将连续的两个 UpdateVertexU 和连续的两个 UpdateVertexV 操作融合成了一个操作。对于 CUBE 系统所使用的三维划分算法（$\mathcal{P}$，L），其中 $\mathcal{P}$ 的选取和 PowerLyra 系统中所使用的二维划分算法是一样的，而 L 则是通过枚举得到的最优值。

#### 3.6.3.2　执行时间

表 3.3 给出了我们测试得到的在各种情况下梯度下降算法和 ALS 算法的一轮的执行时间。其中 D 的取值代表用户和商品的特征向量的大小，一般来说 D 的取值越大最终的预测结果就越精确。和微型测试用例一样，我们同时使用了三个不同的实际用户数据集合 Libimseti，LastFM 和 Netflix 进行了测试。并且同时测量了在使用 8、16、32、64 个计算节点这四种情况下的数据。对于每一种情况，我们都汇报了 PowerGraph、PowerLyra、CUBE 这三个计算系统能够达到的最佳运行速度。对于 CUBE 系统，我们了所有合法的 L 的取值并在表 3.3 中汇报了最佳的 L 取值（括号中的数值）。在表 3.3 中 Failed 表示这一组测试由于内存不够的原因无法运行。

总的来说，从表 3.3 中的数字我们可以看到，CUBE 在梯度下降和 ALS 这两个算法下能够较之 PowerLyra 分别达到最高 4.7 倍和 3.1 倍的加速比。如果比较的是 PowerGraph 系统的话，这一加速比可以进一步的提升到最高 7.3 倍。根据我们的分析，ALS 算法的加速主要的原因来自于我们三维划分算法带来的通讯量减少，而对于梯度下降算法的加速这同时来自于通讯量的减少和我们的矩阵运行后段带来的系统 cache 利用率的提升。这一区别的主要原因在于 ALS 算法的计算部分主要时间花在了对 DSYSV 这一计算密集型的运算进行求解之上，因此通过矩阵执

表 3.3　实际应用的测试结果。表格中每一个单元里列举的数据都采用 “PowerGraph / PowerLyra / CUBE” 的格式，单位为“秒/轮”。

| D | # of workers | Libimseti GD | | | | | ALS | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 64 | 8 | 9.78 | / | 9.56 | / | 2.04 (2) | 70.8 | / | 70.4 | / | 46.7 (8) |
| | 16 | 8.04 | / | 8.16 | / | 1.95 (4) | 72.6 | / | 71.5 | / | 37.6 (16) |
| | 64 | 6.82 | / | 6.89 | / | 2.59 (4) | 87.0 | / | 86.8 | / | 28.7 (64) |
| 128 | 8 | 14.99 | / | 14.94 | / | 3.87 (2) | 261 | / | 258 | / | 193 (8) |
| | 16 | 12.81 | / | 12.91 | / | 2.62 (4) | 270 | / | 270 | / | 135 (16) |
| | 64 | 11.64 | / | 11.62 | / | 3.33 (8) | 331 | / | 331 | / | 109 (64) |
| D | # of workers | LastFM GD | | | | | ALS | | | | |
| 64 | 8 | 12.0 | / | 8.98 | / | 3.45 (2) | 124 | / | 73.5 | / | 70.9 (8) |
| | 16 | 10.5 | / | 8.22 | / | 2.59 (2) | 128 | / | 69.5 | / | 61.6 (16) |
| | 64 | 10.4 | / | 9.86 | / | 2.48 (4) | 158 | / | 111 | / | 57.6 (64) |
| 128 | 8 | 19.0 | / | 13.8 | / | 4.74 (2) | 465 | / | 263 | / | 270 (4) |
| | 16 | 17.6 | / | 13.5 | / | 3.35 (4) | 490 | / | 253 | / | 200 (16) |
| | 64 | 18.6 | / | 17.8 | / | 3.47 (8) | Failed | / | Failed | / | 230 (64) |
| D | # of workers | Netflix GD | | | | | ALS | | | | |
| 64 | 8 | 34.4 | / | 27.7 | / | 6.03 (1) | 256 | / | 204 | / | 110 (2) |
| | 16 | 26.7 | / | 17.3 | / | 3.97 (1) | 186 | / | 107 | / | 60.4 (2) |
| | 64 | 18.3 | / | 7.42 | / | 4.16 (1) | 179 | / | 66.0 | / | 42.5 (8) |
| 128 | 8 | 51.8 | / | 38.6 | / | 9.65 (1) | 865 | / | 657 | / | 463 (1) |
| | 16 | 41.9 | / | 23.0 | / | 6.59 (1) | 669 | / | 340 | / | 258 (2) |
| | 64 | 30.6 | / | 11.3 | / | 6.55 (2) | Failed | / | 239 | / | 118 (8) |

行后端优化局部性并没有很大的用处。在 DSYSV 这一计算核心内部就有足够的内部局部性可以利用。相对来说，梯度下降算法的计算部分非常简单，因此对内存的载入带宽的利用更加敏感。在下一节中我们将更加具体的分析这两部分的构成。

#### 3.6.3.3　通讯量减少

正如我们在前文中提到的，CUBE 较之已有系统的加速比主要的来源有两个方面：1）一个是通过采用三维划分方法替代二维划分方法减少了通讯量；2）另一个是由于采用了基于矩阵的计算引擎从而提升了系统对 cache 的利用率。因此，为了进一步的了解这两部分各自的影响，我们在本节中将计算具体的通讯量减少数量和测试光靠减少通讯量能够达到的加速比。其余部分可以认为是由矩阵计算引擎

带来的。

**GD** 前文第 3.6.2 节中给出的通讯量计算方法可以帮助我们计算各种情况下执行一轮梯度下降算法所需的通讯量。只不过由于我们利用了一个 local combiner 对 UpdateEdge 操作的通讯量进行了进一步的优化，这一步骤的通讯量的实际计算公式为 $2*8\text{byte}*(L-1)*|E|$。同时，由于 Sink 操作只需要进行一轮的 AllToAll 通讯，其通讯量仅为 Push 和 Pull 操作的一半。综上所述，梯度下降算法一轮执行的总通讯量可以由如下公式算出：

$$\begin{aligned}\text{Traffic(GD)} = &(2+2+1)*8\text{byte}*(\lambda_{N/L}-1)*S_C*|V| \\ &+2*8\text{byte}*(L-1)*|E|\end{aligned} \tag{3-7}$$

图 3.9 给出了通过上述公式计算得到的有多少通讯量被减少了的一个更加直观的展示。由于实际的实现是严格按照我们的定义来的，而元数据等的通讯量又远少于实际数据的通讯量，所以实际测试得到的结果也基本上和数学公式计算得到的结果一样。从图 3.9 中我们可以看到，三维划分算法能够造成的通讯量减少与三方面的因素有关：1）图的稠密程度即 $|E|/|V|$；2）向量大小 $S_C$；和 3）二维划分算法的优劣程度 $\lambda$。如果图的稠密程度远远的大于向量大小（即 $|E|/|V| \gg S_C$），则最佳的情况是将所有的计算节点划分到同一层，即设定 L = 1。举例而言，Netflix 数据集的稠密度大于 200，所以从表 3.3 中给出的数据可以看到，在很多情况下 L = 1 下才能达成最好的执行速度（除了 D = 128 和 N = 64 这么一种情况）。相对的，Libimesti 数据集的稠密程度仅为 57，因此我们的三维划分算法可以在这个数据集上达成最高减少 64% 的通讯量的效果。

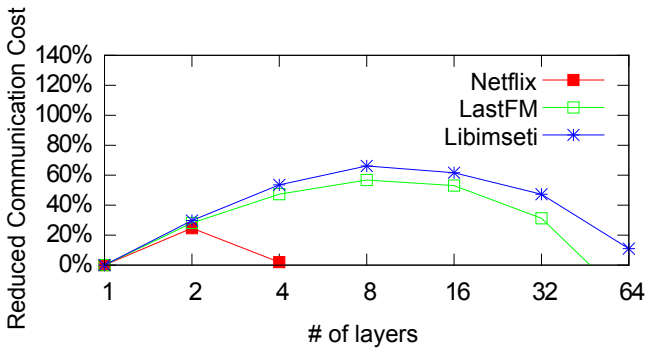



图 3.9　GD 算法在三维划分情况下可以节省的通讯量。

为了进一步的了解这一通讯量的减少具体能够带来多少的加速比，我们还测试了另外的一种情况。具体来说，我们比较了将 L 设定为最优值和将 L 固定为 1

这两种情况的速度。在这种情况下，由于两个算法的执行引擎是一样的，计算任务也都基本是均分的，因此可以认为所有的加速都是来自于通讯量的减少。图 3.10 给出了这一组测试的最终结果。从中我们可以看到，通过三维划分大概能够提升两倍左右的速度，即大概占总的提升的一半。在 LastFM 数据集上的测试结果也显示了基本和在 Libimesti 数据集上的测试结果一样的数字。对于 Netflix 数据集，其最佳的 L 在很多情况下就是 1。因此，在这一数据集上我们大三维划分算法基本上没有什么效果。不过，由于我们通过将用户给出的以点为中心的程序自动地转化成了矩阵计算操作进行执行，我们的 CUBE 系统即使在 Netflix 数据集上也能取得较好的综合性能提升。

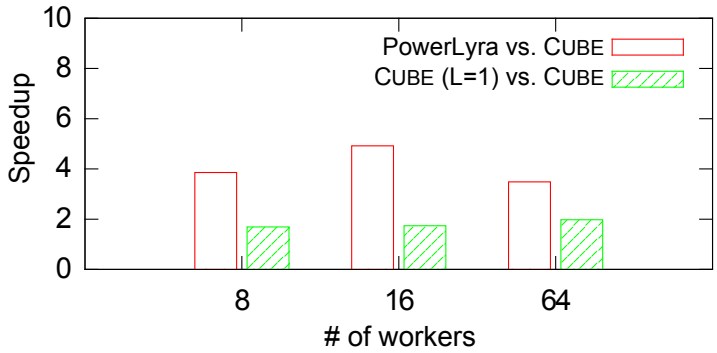



图 3.10　在 Libimesti 数据集上将 L 设定为最优值和将 L 固定为 1 的速度比较。

**ALS** 和前文中描述的一样，我们在实际实现 ALS 算法时将算法 2 中连续的两个 UpdateVertexV 函数和连续的两个 UpdateVertexU 函数都合并成了一个函数去执行。因此，每一轮的 ALS 计算中恰好拥有 UpdateVertexU，Push，UpdateVertexV，Pull 这四个函数各一次，且均为在二部图的特殊情况下自行的。因此，根据前文第 3.6.2 节中给出的公式可以计算得到，每一轮 ALS 算法的执行所需要的通讯量为：

$$\begin{aligned}\text{Traffic(ALS)} =& 2 * 8\text{byte} * (\lambda_{N/L} - 1 + \frac{L-1}{L}) * S_C * (|\mathbb{U}| + |\mathbb{V}|) \\ &+ 8\text{byte} * (L-1) * (|\mathbb{U}| + |\mathbb{V}|)\end{aligned} \tag{3-8}$$

根据上述公式我么可以推导出在三维划分算法（$\mathcal{P}$，L）中的二维划分方法 $\mathcal{P}$ 的效果越差（即 $\lambda$ 的增长曲线越陡）的情况下三维划分方法能够取得通讯量减少越多。

图 3.11 展示了具体的在每一个测试集上变动 L 可以达成的通讯量减少。举例来说，对于 Libimesti 数据集，在使用 hybrid-cut 作为其二维划分方法时把这一数据图切分成 64 张子图所产生的平均副本数 $\lambda_{64} = 11.52$。因此，通过上述公式计算可

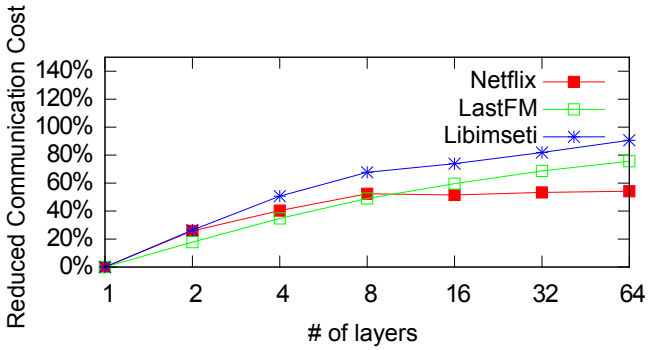



图 3.11　ALS 算法在三维划分情况下可以节省的通讯量。

以得到把这一数据集在一个划分成了 64 层（即 L = 64）的计算集群上执行可以降低 90.6% 的通讯量。相对的，已有的二维划分算法在其他的两个数据集上的效果相对来说要好一些，因此我们的三维划分方法能够达成的通讯量降低比例也就小一点。与梯度下降算法类似，我们同样通过将最佳情况和将 L 固定为 1 的情况进行对比测量了 ALS 算法的综合技术比中有大概多少是由我们的三维划分方法产生的。从结果中我们发现，基本上全部的加速比（>90%）的来源都是我们的三维划分方法。这一现象的主要原因在于 ALS 算法的计算部分主要时间花在了对 DSYSV 这一计算密集型的运算进行求解之上，因此通过矩阵执行后端优化局部性并没有很大的用处。在 DSYSV 这一计算核心内部就有足够的内部局部性可以利用。相对来说，梯度下降算法的计算部分非常简单，因此对内存的载入带宽的利用更加敏感。

### 3.6.4　其他讨论

#### 3.6.4.1　可扩展性

对于大多数的图计算应用来说，随着使用的计算节点数量的增加，其所需的通讯量也逐步增加。因此，这一类算法的可扩展性一般都不算太好，特别是对于那些数据图大小比较小的情况。这一现象的主要原因在于增加的通讯量造成的开销迅速的抹平了减少的计算时间所带来的好处。不过，虽然这一由图计算应用本身性质所决定的扩展性问题无法用其他办法解决，由于我们的三维划分算法说产生的通讯量较之传统方法要少，使用三维划分的图计算系统的可扩展性也比传统的图计算计算框架要好。从表 3.3 中给出的数据我们可以看到，对于 LastFM 和 Libimesti 这两个数据集，PowerLyra 系统的执行时间在将 16 节点的集群扩展到 64 节点时事实上执行时间是增加了的。相对的，我们的 CUBE 系统在大多数情况下可以维持时间下降的趋势一直到 64 个计算节点。当然，不可否认的是 CUBE 系统在系统节点数超过 16 后其可扩展性也相当的受限。不过这主要是由于图计算应用和数据图

大小导致的。对于更大的数据集，比如说工业界所使用的拥有亿万条边的数据集[6]来说，我们的 CUBE 理论上应该是可以扩展到更大的集群上的。为了验证我们的这一观点，我们通过随机生成的方法生成了一张拥有十亿条边且满足 power law 的数据图。通过在其上的测试我们发现，CUBE 系统可以通过将集群大小从 32 个计算节点扩展到 128 个节点获得大约 2.2 倍的加速。同时，由于我们的三维划分方法和其他图计算系统的部件基本上是独立的，因此现有的可以用来优化 PowerGraph 和 PowerLyra 系统的可扩展性的工作[49,50] 也都可以运用于 CUBE 系统。

#### 3.6.4.2　内存开销

从表 3.3 中的数据我们可以发现，对于在 LastFM 和 Libimesti 这两个数据集上运行 ALS 这个算法来说，其最佳的执行时间大多数的情况下都是在将层数 L 设置成和集群中所包含的计算节点数 N 一样的情况下获得的。然而，虽然一直将 L 设置成 N 可以在效率上达到最优值，但这不意味着这一设置在所有的指标上都是最优值。拿内存开销来说的话，通过增大 L，我们减少了所需副本的数量，因此总共的 DColle 数据的数据量减少了。但是相反的，由于每一层的计算节点都需要保存一份完整的 DShare 数据的拷贝，这一部分的数据量是与 L 成正比的。拿 ALS 这个算法举例的话，由于每一条边的 DShare 都是一个大小为 8byte 的 double 变量，其总体的内存开销可以由 $(\lambda_{N/L} * S_C * |V| + L * |E|) * 8$ 这个公式计算得到，其中向量大小 $S_C$ 的值为 $D + D * D$。

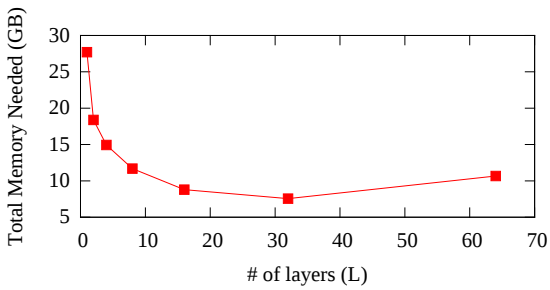



图 3.12　ALS 算法的总内存开销。

图 3.12 展示了通过这一公式计算得到的在不同的 L 的设置情况下总体的内存开销。从上图中我们可以看到，总体的内存开销先是随着 L 的增大逐渐减少。但是，在越过某一临界值之后（大约是 32），其变化趋势逆转，开始随着 L 的增大逐渐增大。这一现象的主要原因在于 DShare 数据的增加已经达到 DColle 的数据的减少所不能中合的地步了。因此我们可以看到，虽然将 L 设定为最高的 N 可以减少执行时间，但是是在增加中的内存开销的代价下的。只不过，由于即便是这种

情况下我们的中的内存开销还是比将 L 设定为 1 的情况下要小得多，所以总体上来说三维划分方法不仅可以降低通讯量，还可以减少内存开销。

#### 3.6.4.3　划分时间

有一部分的调研工作[51] 指出如果使用非常复杂的划分算法的话可能会导致划分算法本身的开销巨大以至于无法实用化的结果。不过，根据最新的调研[11]，这一现象仅仅会在比较复杂的划分算法上成立。相对的，对于基于启发式方法的简单划分算法，其划分开销一般都不大。对于我们的三维划分方法（$\mathcal{P}$，L）来说，其划分的开销读基本上完全取决于起所基于的二维划分方法 $\mathcal{P}$ 的划分复杂度。因此只要选取比较简单的二维划分算法作为 $\mathcal{P}$，我们的三维划分方法的划分复杂度也就可以做到比较低的程度。同时，由于会利用到三维划分的图计算应用一般来说每一轮执行所需的通讯量都比较大，所以在我们的测试中没有发现划分时间超过一轮迭代时间的情况。这种程度的划分开销显然是可以接受的。

具体来说，整个的 Cube 系统的初始化过程分为以下的三个阶段。第一个阶段是加载阶段，其中集群里的每一个计算节点分别加载不相交的一部分数据图中的数据。由于这一部分和已有系统是完全一样的，所以开销上也是完全一样的。第二部分是划分，在这里因为我们使用的都是和 PowerLyra 系统所使用的一样的二维划分算法。而这一划分结果又可以被不同的层共享，只需要计算一次。所以总体上来说这一阶段的开销和 PowerLyra 系统也是一样的。最后一部分为数据的再分配，需要将原来由某一个节点加载但计算之后发现并不由这个节点管理的数据发送给相应的计算节点。在这一阶段，如果没有 DShare 的权值，或者其权值都是随机初始化的话，由于最终 DColle 的数据量会随着 L 的增大而减少，其在分配的开销事实上是比传统的开销还要小的。在有 DShare 的给定初始值的情况下，这一部分的开销的变化曲线基本上和图 3.12 中所给出的中内存开销的值相当，因此也是一个先减少再增加的趋势。不过总体上来说其通讯量较之直接使用二维划分方法来说还是要小的。

#### 3.6.4.4　COST

一个近期的调研[24] 发现许多的图计算系统之所以能够拥有良好的可扩展性主要是因为他们的单线程时间的效率不高。为了避免这一问题，他们提出了一个新的测量指标“超过优化后的单线程程序所需的计算节点数”，简称为 COS）。如果一个系统的 COST 值为 c，则说明这个系统在扩展到 c 个计算节点之后它的运行速度就比一个优化后的单线程程序要快了。为了测量我们 Cube 系统的 COST 值，

我们手工实现了梯度下降和 ALS 算法的单线程程序。其具体实现方法基本上就是前文第 3.4.6 节中所给出的数学公式直接翻译成 BLAS 或者 LAPACK 操作。通过测试，我们发现我们的 CUBE 系统的 COST 值大约为 4，属于可以接受的范围。相对的，在提出 COST 这一指标的论文中表示“目前很多的图计算系统的 COST 值都超过一百”。我们的这一结果与一些其他的关于矩阵执行引擎执行效率的测评工作[15,52] 是一致的。矩阵执行引擎的 COST 只一般来说都要远好于普通的基于点的执行引擎。

#### 3.6.4.5　更快的网络

我们的测试平台所用的互联网络是 1Gb 的以太网。虽然这是一个非常常见的配置，但随着网络技术的不断发展，目前有很多更快的网络设备也逐渐的被投入到使用之中。因此读者客人想要知道在更快的网络配置之下我们的三维划分算法是否依然可以获得可观的加速比。然而，事实上根据我们的测试在 10Gb 的网络环境下 CUBE 系统相较于 PowerGraph 和 PowerLyra 系统的加速比反而变得更大了。这一反常现象的主要原因在于 PowerGraph 和 PowerLyra 系统对高速网络的利用率不高，以至于没有完全享受到这一网络带来的好处。相对的，我们的 CUBE 系统是基于 MPI 的，其通讯时间在 10 倍的网络带宽环境下基本上就精确地减少了 10 被的执行时间。

#### 3.6.4.6　图结构的影响

正如我们前文所提到的，图的结构特性对我们所提出的新型划分算法的效果有着很大的影响。总的来说，越是倾斜度小（less skew），即分布均匀的图数据传统的二维划分算法能够利用的特质就越少。因此他们往往在这一类的图上表现较差。相对的，我们的三维划分方法在平均副本数 $\lambda$ 增长越快的图，即传统二维划分方法表现较差的图上能够取得的加速比越高。因此，为了屏蔽选取了一个不恰当的二维划分算法导致我们的测试结果不准确的情况，我们枚举了所有提供的二维划分算法，并选用了其中使得 $\lambda$ 的增长曲线最平缓的那一个。

#### 3.6.4.7　应用范围

上文中所提到的图计算的应用仅仅为我们的划分算法能够应用的范围的举例。在没有提到的算法中最常见的一类是利用到了前文第 3.6.2.1 节中的微型测试用例 SpMM 的算法。这一操作由于其普遍性被很多的机器学习和数据挖掘所利用。除此之外，还存在很多的算法他们的简单模式并不具备可划分的点权和边权，但是

存在对应的复杂模式拥有可划分的点权和边权比如说多源最短路径[53] 对应多源最短路径。虽然前者的点权不可划分，但是后者的点权是可以的。

## 3.7　本章小结

总的来说，本章中所描述的三维划分方法的原理来自于这样一个发现：对于很多种类的图计算应用，其任务划分等价于图划分的假设并不成立。对于这些问题来说，一个点或边的属性是一个由数据元素组成的向量，而不是一个单独的数值。利用这个隐藏的维度，我们可以将一个点的属性向量进行划分并分配给不同的节点计算。基于这个发现，我们主要做出了以下几个方面的贡献：1）我们提出了一种新颖的、在分布式图处理系统中对图进行三维数据划分的方法。2）我们提出了一种针对三维划分设计的新的编程模型 UPPS。3）基于三维划分和 UPPS 编程模型，我们实现了一个分布式的图处理引擎 Cube。4）我们系统验证了三维数据划分应用在分布式图处理中对性能的改进。实验表明，Cube 的效率显著优于 PowerLyra 和 PowerGraph。

# 第 4 章　外存图计算系统的分层数据组织

## 4.1　本章概述

前文第 2.2 节中有提到，当前除分布式图计算系统之外的另外一个大的大规模图计算处理系统类别为外存图计算系统[12–14,28]。这一类系统通过只在内存中存储当前所需的部分图计算数据，而将其余部分的图数据存储在外存之上的方式极大地扩展了它们的数据容量。举例而言，著名的开源图数据集中最大的 Yahoo 网页链接数据集[27] 大约拥有 14 亿个点和 66 亿条边。在计算 PageRank 这一图计算应用时，系统所需处理的总数据量大约为 14B*8byte + 66B*16byte = 116.8GB。显然可以发现，虽然这个规模的数据量很少有单台机器的内存可以完全容纳，但是绝大多数主流 PC 机的外存设备容量都远高于它。因此，可以认为，通过借用外存来突破单机内存的容量限制可以使得单机图计算系统也能够处理相当规模的图数据集。在这里，我们假设每一个点的点权为一个 double 型变量，占用 8 byte 空间。每一条边的边权也是一个 double 变量，不过由于还需存储于这条边两个点的标号所以一共占用了 16 byte 的空间。这两个数字会随着图计算应用种类的不同而不断变化，但是总体上来说通过使用外存可以极大地扩展图计算系统的处理容量的这一趋势不变。

与分布式图计算系统的情况类似，虽然随着技术的发展外存设备的速度也在不断提升，但相对于内存速度来说依然是相差了一个甚至数个数量级。尤其是在读取的数据内容顺序性不好的情况下，其差距会非常地明显。因此，外存图计算系统的数据载入路径之上的第一个阶段，即从外存将数据读入到内存之中去这一阶段，经常成为外存图计算系统整体运行速度的瓶颈。而为了提升这一步骤上的速度，已有的外存图计算系统所采用的方式主要在于降低对外存设备访问的随机性[12–14]。从前文第 2.2.1 节给出的表 2.1 中所列的数据我们可以看到，当程序仅仅以 1KB 为区块大小进行随机读写时，SSD 的带宽仅仅只有 6MB/s，这显然是无法使用的。相对的，如果使用特定的方法将每次读取的数据区块大小扩大到 64MB，同样的一块 SSD 就可以提供大约 772MB/s 的带宽。由于这两个速度上显著的差距，不难理解为什么已有的外存图计算系统都着重于解决随机性的问题。

但是，事实上，外存图计算系统从外存将所有所需数据载入到内存中去这一阶段所消耗的总时间的公式为“总载入数据量/平均载入带宽”。因此，显然可以发现，决定这一最终值的因素实际上是有两个的。除了通过提升外存访问的顺序性以增加平均访问带宽的形式之外，减少总载入数据量也可以达到同样的效果。对

于传统的单机全内存图计算系统来说，其总载入数据量是固定的，即为总的点数据大小和总的边数据大小之和。但是，由于单机内存容量的限制，外存图数据系统一次只能载入一部分的图数据，即一张原图的子图。在这种情况下，不可避免的有一部分的数据会被需要重复的载入以保证最终计算结果的正确性。而且，通过分析可以发现，一般来说一次性载入的点的数量越多，每个点平均需要被载入的次数就越少。因此，为了达成减少总载入数据量的目的，一个可行的方法就是增加一次可以载入的点的数量。

不过，和分布式图计算系统的情况类似，传统的外存图计算系统也都假定一个点的点权是不可被进一步划分的对象。在这种情况下，如果给定最高能够提供的内存容量 M，其所能借容纳的最大点数也是确定的，即 M/sizeof(vertex)。但是，正如我们在前文第 3.2 中举例说明的，当前有很多的数据挖掘和机器学习问题也被建模成图计算问题求解。而对于这一类问题来说，经常出现每一个点的点权是一个向量，因此可以被进一步划分等情况。所以，我们可以通过将点划分成子点的方式将每个点的大小从 sizeof(vertex) 减少到 sizeof(vertex)/L，其中 L 为划分成的子点的数量。这也就意味着我们可以载入 L 倍于原来的点数，从而减少每个点被重复载入的次数。

当然，这样的方法并不是有百利而无一害的。在后文中我们会通过分析展示，虽然通过分层的方式可以减少点数据的载入，但相对的它也会增加边数据的载入。因此，这意味着 L 是一个需要综合考虑的变量。不过，根据我们的测试结果，通过选取一个合适的 L 的方式，在很多情况下我们都可以达成远低于原来方式的总数据载入量。为了进一步验证我们的这一结论，我们基于已有的外存图计算系统 GirdGraph 设计了支持分层存储的外存图计算系统 3DGridGraph。通过测试我们发现，3DGridGraph 较之 GirdGraph[14] 最高可以达到 4 倍的加速。

## 4.2　背景介绍

与分布式图计算系统中的图数据划分算法类似，对于外存图计算系统中的外存图数据组织格式，我们也可以将其分为一维和二维两种主要的类别。在这一节中，我们将分别对这两类外存图数据组织格式中的代表系统，即 GraphChi[12] 和 GridGraph[14] 进行介绍。

### 4.2.1　外存图计算系统中的一维划分：GraphChi

在前文的第 2.2.2 节之中我们已经简略的介绍了 GraphChi 外存图计算系统的基本情况。这是一个以点为中心的外存图计算框架，因此也可以被称之为是一维

的外存图计算系统。在 GraphChi 中，每一次计算的单元为一个点，所需的数据为这个点的点权和与其相关的所有边的边权。这种情况下，由于一次只访问一个点的点权，所以对点权的访问很容易就可以做到顺序访问。相对的，在 GraphChi 系统中，边权如何组织是至关重要的问题。由于图计算应用本身局部性不好的特点，如果只是简单地按照源点或者汇点对边权进行排序的话：每一次访问一个点时，要么写入其出边数据的顺序性很好但是读取入边数据的局部性很差；或者读取其入边的顺序性很好，但是写入其出边数据的顺序性很差。

为了解决这一问题，前文的 2.2.2 节中我们有提到 GraphChi 巧妙地将边数据组织成了 Parallel Sliding Windows 的格式。具体来说，它先将图中的点分为多个连续的 interval，每个 interval 对应一个边的 shard 存储这一 interval 中所有点的入边。而在每一个 shard 之中，边又是按照它们的源点进行排序的。在这种情况下，每一次计算一个 interval 中所有点对应的点程序时所需访问的数据就可以用几次大块的连续读取完成。

具体来说，为了计算第一个 interval 中的所有点相关的点程序，需要载入这个 interval 中所有点的入边和出边。其中，入边全部存储在对应的第一个 shard 之中，因此可以一次性的全部读取进来。相对的，出边由于被分配在不同的 shard 之中所以需要分别读取。但是，由于在每个边的 shard 之中边的顺序都是按照源点进行排列的，所以可以想象，在每一个边的 shard 之中，其与第一个 interval 对应的出边的数据是连续存放在这个 shard 的开始部分的，因此可以一次性读出而不用进行随机访问。由于数据的读取方式是入边数据一次读取每一个 shard，出边数据每次在所有的 shard 中从上到小读取连续的一块，所以每一次 interval 的计算都只进行了 interval 个数次的顺序读写，完全避免了小块的随机读写。

同理，上述的情况还可以推广到任意一个 interval 之上。对于任意一个 interval i，其相关的入边数据都存储在 shard i 之中。同时，虽然其相关的出边数据同时存在于各个 shard 之中，但必然是每个 shard 中连续的一部分。因此，对于一张被分为了 $P$ 个 interval 的图，每次计算一个 interval 时只需要进行 $P$ 次顺序读写，即每一轮迭代总共进行 $P^2$ 次顺序读写。通过将对外存的访问组织成这样的形式，GraphChi 系统极大地提升了外存图计算系统的执行效率。根据它们自己的测试，一个在普通 PC 上运行的 GraphChi 系统在特定情况下甚至可以拥有与大约 100 个核的分布式计算集群相当的计算效率。

### 4.2.2　外存图计算系统中的二维划分：GridGraph

与一维的外存计算系统相对，GridGraph 是一个二维的外存图计算系统，其每次计算的单元是一条边。换句话说，GridGraph 每一次计算仅仅需要读写一条边的边权以及这条边对应的源点和汇点的点权。这一变迁在本质上和分布式环境下的以点为中心的一维编程模型“点程序”变成了以边为计算中心的二维编程模型 GAS[3] 是一样的。

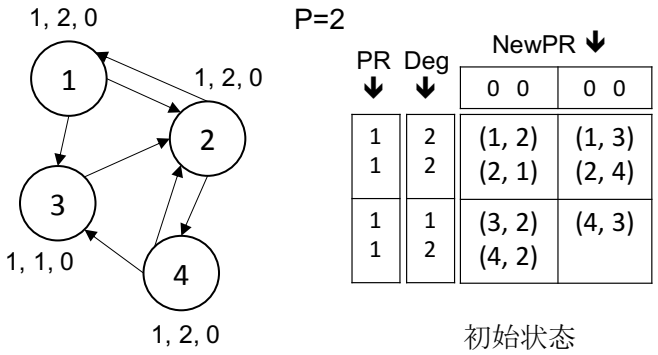



图 4.1　用 GridGraph 系统计算 PageRank 时的初始状态[14]。

具体来说，在 GridGraph 系统中数据图的点集也会被分为 P 个连续的 interval。不过，和 GraphChi 系统将边集对应地一维切分成 P 个 shard 不同，GridGraph 系统实际上将整个边集合二维切分成了 $P^2$ 个集合（被称之为 grid）。如图 4.1 所示，在 P=2 的情况下，原图的所有 4 个点被分成了两个 interval。其中第一个包含点 1 和点 2，第二个 interval 包含剩下的两个点。与此同时，这一数据图的边集合被分为了 2*2=4 个 grid，其中 grid[i][j] 包含从 interval i 中的点指向 interval j 中的点的边的集合。举例来说，图 4.1 中的 grid[0][1] 就包含了从点 1 或者点 2 指向点 3 或者点 4 的边，即 (1,3) 和 (2,4)。

在计算 PageRank 这一图计算应用的过程中，每一个点的点权包含三个部分：（1）一个是 PR，即当前这个点的 page rank 值；（2）一个是 Deg，即这个点的出度大小；（3）最后一个是 NewPR，即这一轮迭代计算出的新的 page rank 值，初始时均为 0。在图中我们可以看到，这几个不同的点权分别由一个一维的向量存储，并且也各自切分成了两个 interval。只不过，由于在计算的过程中 PR 和 Deg 这两个点权是只读的，所以其位置位于边数据矩阵的左侧，意味着它和边的源点是相关连的。相对的，因为 NewPR 这个点权是只写的，因此它被放在数据矩阵的上方，意味着它和边的汇点是相关连的。在实际的运算过程中，每当 GridGraph 加载一个 grid 的边数据 grid[i][j] 时，其系统会自动的读取对应矩阵左侧的所有点权数据的第 i 个 interval 用于计算，而结果会写到对应矩阵上方点权数据中的第 j 个

interval 中去。在这一个 PageRank 的例子中，对每一条边 (u,v) 执行的计算过程为 $NewPR[v] += PR[u]/Deg[u]$。

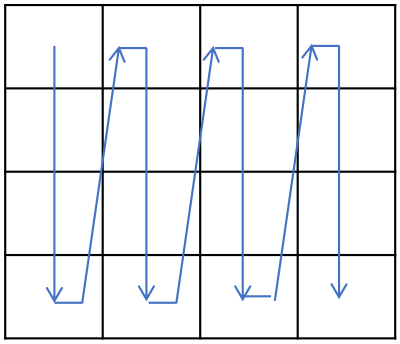


图 4.2　当 P=4 时 GridGraph 系统中 grid 的处理顺序。

通过将数据组织成这样的形式，GridGraph 同样可以做到将原本对图数据的随机访问组织成数次连续的大块外存访问的形式。只不过，GraphChi 中每一轮的迭代分为 $P$ 个阶段，每个阶段都是处理一个 intervel 内的点相关所有数据，需要一次连续的点权读取和 $P$ 次连续的边权读取。相对的，GridGraph 中每一轮的迭代被分为了 $P^2$ 个阶段，每个阶段仅仅需要处理一个 grid 的边数据，对应的点权也都是连续存放在一起的，因此每个阶段仅需要一次连续的点权读取和一次连续的边权读取。同时，GridGraph 并不会按照随机的顺序处理 grid 而是按照如图 4.2 所示的顺序进行处理的。因此，实际上 GridGraph 可以以流的形式连续地读取边的数据和写入点的数据（NewPR），只不过在点的数据的读取上（即读取 PR 和 Deg）需要折返 $P$ 次。显然，这样的存储格式在局部性上较之 GraphChi 每一轮迭代需要 $P^2$ 次连续的大块读取的形式更具优势。

同时，除上述顺序性更好的优点之外，GridGraph 还有其他的两个好处：（1）一个是 GridGraph 对点权的更新是可以立即生效的。因此，在一轮迭代的过程中处理顺序上较后的 grid 时可以读取到处理顺序较前的 grid 时已经更新过了的数据。对于迭代类的图计算应用，这样的特性可以加快其收敛速度，从而减少整体的运行时间；（2）同时，GridGraph 还实现了一个非常简单但也非常有效的选择性载入功能。具体来说，如果过 interval i 对应的点权数据没有被更新的话，在这一轮迭代中 GridGraph 会选择性地略过所有满足形式 gird[i][*] 的 grid 中的边权数据。这一方法近似的模拟了分布式图计算系统中设定一个点是否为“活动的（active）点”的功能，从而降低了部分情况下图计算应用所需载入的总数据量。根据他们的测试，GridGraph 较之更早的 GraphChi 和 X-Stream 等外存图计算系统都有着明显地效率提升。

## 4.3　3DGridGraph

在前一节中，我们描述了已有外存图计算系统对其图数据的组织形式。从中我们可以发现，和传统的分布式图计算系统一样，已有的外存图计算系统也都将数据图中的点当成是不可被进一步划分的单元。这一假设在最早的一维划分情况（即 GraphChi）下是无关大局的，因为无论 GraphChi 将数据图中的点划分成几个 interval，其每一轮迭代都仅仅会读取每个点一次。但是，在二维划分的 GridGraph 系统中，如果一张图的点被划分成了 $P$ 个 interval，那么每一点都会被读取 $P$ 次，因此我们自然的会希望尽可能地减少 $P$ 的大小。基于这一目的，我们设计了分层的图数据组织格式，并通过它将 GridGraph 系统扩展成 3DGridGraph 系统。在本节中，我们将首先描述这一组织格式和为什么它能够减少外存数据的读取量。随后，我们将定义一个与之相配合的编程模型并用具体的算法实例展示其用法。

### 4.3.1　分层存储优势

#### 4.3.1.1　分层前的情况

对于 GridGraph 系统来说，如果给定它最多允许使用的内存空间大小为 $M$ 的话，事实上其可以允许的最小的 $P$ 的数值是确定的。具体来说，由于在 GridGraph 系统的执行阶段边的数据是可以以流的形式读取的，因此理论上并不需要对边的数据进行事先地缓冲区分配。但是，相对来说，由于在访问一个 gird 所包含的边的数据时对应需要访问的点的数据在一定范围内是随机的，因此需要将它们放置在内存之中以避免对外存的随机读写。即便是不考虑多线程并行执行的情况，GridGraph 系统最小处理的单元至少要是一个 grid，即 grid[i][j]。此时，必须要在内存里分配包含在 interval i 和 interval j 的点的点权缓冲区。在假设数据图中点集大小为 $|V|$，每个点权大小为 $S_V$，且 i 不等于 j 的情况下，需要占用 $2*S_V*|V|/P$ 大小的内存。所以，为了保证这一过程的正常运行，需要 $M \geq 2*S_V*|V|/P$ 即 $P \leq 2*S_V*|V|/M$。换句话说 $P$ 的最小取值为 $2*\lceil S_V*|V|/M \rceil$。

在给定 $P$ 之后，参考图 4.2 中给出的 GridGraph 系统对各个 gird 的访问顺序，我们可以通过余下的步骤计算出 GridGraph 执行一轮迭代所需对外存读取和写入的数据总量。首先，由于边的数据是以流的形式读取的，且在 GridGraph 系统中不允许对边的数据进行修改，所以这一部分无论 $P$ 的取值为何都是一共需要从外存读取 $S_E*|E|$ 大小的数据，其中 $S_E$ 为每条边的大小 $|E|$ 为边的总数。同样的，由于使用了如图 4.2 中所示的运行顺序，每一个 interval 的点权数据仅仅需要写出一次。所以，无论 $P$ 的取值为何，GridGraph 中对外存的写入也都固定为 $S_V*|V|$。不过，为了达成上述目的，结果就是每一个点的数据都需要被读取 $P$ 次。所以总体上来

说，GridGraph 系统一轮迭代需要读取的外存数据总量为 $S_E * |E| + P * S_V * |V|$。在给定内存上限 $M$ 的情况下，上述公式能够取得的最小值为：

$$\text{Read(M)} = S_E * |E| + 2 * \lceil S_V * |V| / M \rceil * S_V * |V| \tag{4-1}$$

#### 4.3.1.2　分层后的情况

上一节中我们计算的是在分层前，即假定数据图中点权不可被划分的情况下所能达成的最小数据读取量。但是，正如我们在前文第 3.2 节中提到的，事实上，目前有很多的数据挖掘和机器学习算法也被建模成图计算应用计算。在这一类的图计算应用当中，有很多的应用定义的每个点的点权为一个可以被进一步划分的向量。因此，参考我们在前文第 3.3 中提出的三维数据划分的基本思想，同样的我们也可以在外存图计算系统中将一个点的点权拆分成 $L$ 份。考虑到后续需要进行的计算大多都是对位地计算，在这一拆分的过程中，同样应该将点权中向量下标相同的元素归到同一份中去。同时，考虑到不整除的情况，可以认为这 $L$ 份中最大的那一份大小为 $\lceil S_C / L \rceil$。

显然的，在同样使用不超过 $M$ 大小的内存的情况下，由于我们减少了每一份子点权的大小理论上可以使用的最大的 $P$ 增加成了 $2 * \lceil \lceil S_C / L \rceil * |V| / M \rceil$。同样的，如果该图计算应用仅仅对点权向量进行对位地操作的话，就没有必要同时读取多个层的点权。此时，对点权的读取依然是需要 $P * S_V * |V|$，不过由于 $P$ 地大小减少了，所以这一数值也随之减少了。但是，为了达成这一目的，代价是原本只需要读取一次的边数据现在需要每层读取一次，即一共读取 $L$ 次。综上所述，在给定内存使用上限 $M$ 和层数 $L$ 的情况下，3DGridGraph 系统执行一轮迭代所需的外存总读取量为：

$$\text{Read(M,L)} = L * S_E * |E| + 2 * \lceil \lceil S_C / L \rceil * |V| / M \rceil * S_V * |V| \tag{4-2}$$

显然，这一公式的前半部分是与 $L$ 的大小成正比的。但是，其后半部分是与 $L$ 呈负相关的。在某些情况下（如 SpMM 和 ALS 算法中），$S_V$ 地大小会远大于 $S_E$，此时公式 4-2 的后半部分起主导作用，即增加 $L$ 会带来整体读取量的下降。在后文的第 4.4.1 节中我们会用具体的数据详细描述这一 $L$ 的取值所带来的结果。总体上来说，程序要需要选定一个合理的 $L$ 使之平衡两方面的结果，最终产生最好的

表 4.1　3DGridGraph 所使用的删减版 UPPS 编程模型。

| **Data** | |
|---|---|
| $G$ | — $\{V, E, D = \{DShare, DColle\}, S_C\}$ |
| $G_{bipartite}$ | — $\{\mathbb{U}, \mathbb{V}, E, D = \{DShare, DColle\}, S_C\}$ |
| $DShare_u$ | — a single variable |
| $DShare_{u\to v}$ | — a single variable |
| $DColle_u$ | — a vector of variable with size $S_C$ |
| $DColle_u[i]$ | — the $i^{th}$ element of $DColle_u$ |
| $D_u[i]$ | — abbreviation of $\{DShare_u, DColle_u[i]\}$ |
| $D_{u\to v}[i]$ | — abbreviation of $\{DShare_{u\to v}\}$ |
| **Computation** | |
| *UpdateVertex*($\mathcal{F}$) | — **foreach** vertex $u \in V$ **do**<br>$D_u^{new} := \mathcal{F}(D_u)$; |
| *Push*($\mathcal{U}$) | — **foreach** vertex $v \in V$, index $i \in [0, S_C)$ **do**<br>**foreach** $(u, v) \in E$ **do**<br>$D_v[i] := U(D_u[i],\ D_{u\to v}[i],\ D_v[i])$; |
| *Pull*($\mathcal{U}$) | — **foreach** vertex $u \in V$, index $i \in [0, S_C)$ **do**<br>**foreach** $(u, v) \in E$ **do**<br>$D_u[i] := \mathcal{U}(D_v[i],\ D_{u\to v}[i],\ D_u[i])$; |

效果。

### 4.3.2　编程模型

在前文第 3.4 中我们有提到，已有的以点或者以边为中心的编程模型都假定一个数据图中同一个点的所有权值都是放在一起的，因此可以很廉价的获取到。由于这一假设在进行三维划分的情况下是不合理的，所以我们设计了新型的 UPPS 模型与我们的三维划分算法相配合。同样的，由于在本文提出的 3DGridGraph 系统中我们试图通过将一个点的点权分开存放的方式减少总体的外存数据读取量，原有的 GridGraph 的编程模型也不再适用了。因此，我们同样提出了新的编程接口与之适配。

如表 4.1 所示，事实上我们为 3DGridGraph 设计的编程接口就是原有的 UPPS 模型的一个简化版本。在数据模型方面，我们依然是使用的数据图模型并且要求用

户将权值数据显式地分割成不可再分的 DShare 和可再分的 DColle 两个部分。不过，在 3DGridGraph 系统中，只有点权数据需要被分成上述的两个部分，相对的边权数据仅有 DShare 部分。这是因为 GridGraph 并不支持对边权的修改所以在我们调查的范围内可以用 GridGraph 系统实现的算法都不需要边权的 DColle 部分。同样的，由于边的数据是不可修改的，所以在计算部分我们删去了对应的 *UpdateEdge* 和 *Sink* 这两个函数。此外，由于 3DGridGraph 是一个单机系统，其可以立即访问每一条边 (u,v) 对应的两个点的权值，因此在执行 *Push* 和 *Pull* 操作的过程中它可以直接操作对应点的权值，而不需要用一个中继的 update 进行传递。

### 4.3.3　实例研究

为了展示 3DGridGraph 系统的使用方法，我们同样使用前文第 3 章提到的两个常见的算法 SpMM 和 ALS 作为例子。对于 SpMM 算法来说，其进行的计算完全都是对位的操作，因此只需要一个 *Push* 操作就可以完成。相对的，算法 3 展示了 ALS 算法在 3DGridGraph 系统中的实现。从中我们可以发现，在 3DGridGraph 下进行编程基本上和在 CUBE 上用 UPPS 模型编程是一样的。并且由于单机操作的简便性我们不需要一个边的 DColle 部分作为中间结果的缓存区进行中转，可以直接将 *Vec* 和 $Vec^t * Vec$ 部分的结果直接累和到对应点上计算 $\sum_{(u,v)\in R} R_{u,v} P_u^T$ 和 $\sum_{(u,v)\in R} P_u^T P_u$。在完成累和之后就可以简单的用一次 *UpdateVertexV* 或者 *UpdateVertexU* 操作完成最终的更新了。这里由于具体算法我们已经在前文第 3.4.6 节中进行了详细描述，因此这里就不再赘述。

### 4.3.4　实现

由于 3DGridGraph 是一个单机系统，其对于我们给出的编程模型的实现事实上是非常简单的。具体来说，*UpdateVertex* 函数的实现只需依次遍历各个点即可。而对于 *Push* 这个函数，其执行过程基本上也和原来 GirdGraph 系统的执行过程是一摸一样的，只不过需要对每一层的数据都分别进行计算。在每一层内，3DGridGraph 也都是按照图 4.2 所示的顺序对 gird 进行访问的。在访问各个 gird 的期间，系统会分别执行每条边对应的 Update 函数 $\mathcal{U}$。与 *Push* 函数类似，*Pull* 函数也需要每层执行一次遍历所有 gird 的操作。只不过，为了保证每次更新的数据只需要统一进行一次写入操作，*Pull* 函数遍历 gird 的顺序是按行遍历的，即如图 4.3 所展示的顺序。

**算法 3** ALS 算法在 3DGridGraph 系统中的实现。

**Data:**

$S_C :— D + D * D$

$DShare_u :—$ NULL;　$DShare_{u \to v} :— \{double\ Rate\}$

**Functions:**

```
F1(u_i, e, v_i) :— {
    if i < D do
        v_i.DColle[i] += e.DShare.Rate * u_i.DColle[i];
    else
        v_i.DColle[i] += u_i.DColle[i];
    return v_i;
}
F2(v) :— {
    DSYSV(D, &v.DColle[0], &v.DColle[D]); return v;
    foreach (i, j) from (0,0) to (D-1, D-1) do
        v.DColle[D + i * D + j] := v.DColle[i] * v.DColle[j];
    return v;
}
```

**Computation for each iteration:**

$Push(F_1)$;

$UpdateVertexV(F_2)$;

$Pull(F_1, F_2)$;

$UpdateVertexU(F_2)$;

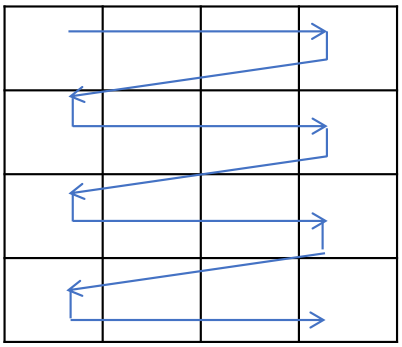


图 4.3　当 P=4 时 3DGridGraph 系统执行 *Pull* 函数中 grid 的处理顺序。

## 4.4　测试结果

### 4.4.1　定量分析

通过类似前文第 4.3.1.2 节中给出的分析手段我们可以推导出，在给定 $P$ 和 $L$ 的情况下，3DGridGraph 系统执行一次 *Push* 或者 *Pull* 操作所需读取和写入的外存

数据总量为：

$$\text{Traffic(P,L)} = L * S_E * |E| + (P+1) * S_C * S_V * |V| \tag{4-3}$$

这一公式中前半部分意味着我们一共读取了 $L$ 次边的数据（$S_E * |E|$），而后半部分则表示我们重复读取了 $P$ 次点的数据（$S_V * |V|$）并写入了一次。在这种情况下，所必须使用的内存大小 $M$ 为 $2 * \lceil S_V/L \rceil * |V|/P$。从这个公式中我们可以发现，如果我们定义 $N = P * L$ 的话，基本上只需要 $N$ 的数值是一定的那么所需的 $M$ 的数值就是一定的。

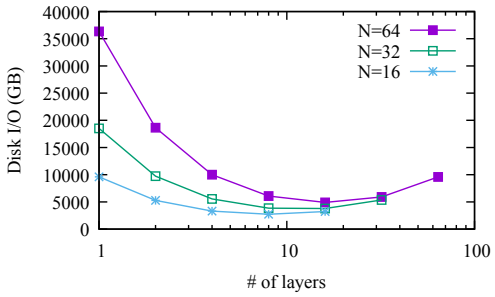



图 4.4　在 Last.fm 数据集上计算 $S_C$=128 的 SpMM 应用时一次 *Push* 或者 *Pull* 操作所需读取和写入的外存数据总量。

因此在上图 4.4 中，我们分别展示了不同情况下固定 $N$ 变化 $L$ 所带来的最终数据读写量。其中为了尽可能地接近现实结果，我们对 $|V|$ 和 $|E|$ 这两个值的使用固定为 Last.fm 数据集的实际大小，即 $|V| = 570416$，$|E| = 17359346$。同时，对于 $S_E$ 和 $S_V$ 这两个值的使用我们同样给出的是在运行 SpMM 算法中常见的一种配置，即 $S_E = 8$ byte，$S_V = 128 * 8 = 1024$ byte 的情况。此时，SpMM 中较小的那个矩阵中较小的那一维长度为 128。有图中的数据我们可以看到，首先随着 $N$ 的数量的增大即所使用的内存数量的下降，系统整体运行所需的外存读写数据总量是在不断增大的。但同时，通过在固定 $N$ 的情况下（即固定可使用的内存大小 $M$ 的情况下）增加 $L$，所需的外存变化趋势是先降后升。这也与我们在前文中的分析是一致的。总体上来说通过对数据进行分层存储，在图 4.4 所展示的情况下，我们最多可以降低 $3.51\times \sim 7.41\times$ 倍的外存读写。

### 4.4.2　实测结果

除上节所示的定量分析之外，我们还基于我们实现的 3DGridGraph 系统实际测量了分层数据存储的加速效果。图 4.5 所示的是执行 ALS 算法时使用不同的层

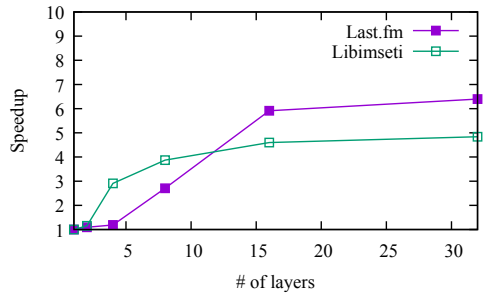



图 4.5　在 Last.fm 和 Libimseti 数据集上计算 $D = 64$ 即 $S_C = 4160$ 的 ALS 算法时不同层数 $L$ 下的加速比。初始二维划分为 32*32 的 grid，因此最高 $L$ 可以取到 32。

数分别产生地归一化后的加速比（以只分一层，即二维划分时为 1，初始二维划分为 32*32 个 grid）。从图中我们可以看到，随着层数的不断增加我们的加速比也在不断地提升。在两个不同的数据集上最高都可以达到大约 6.4 倍的加速。由此可见，我们的是优化方法是十分有效的。

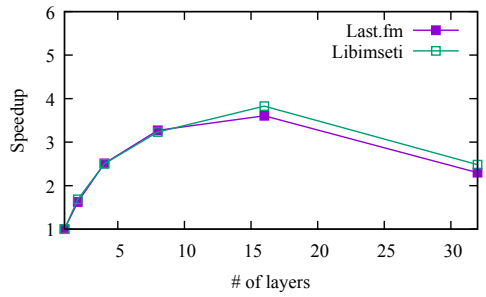



图 4.6　在 Last.fm 数据集上计算 $D = 16$ 即 $S_C = 272$ 的 ALS 算法时不同层数 $L$ 下的加速比。初始二维划分为 32*32 的 grid，因此最高 $L$ 可以取到 32。

但是，虽然在上述的例子中并没有得到展示，并不是所有的情况都是将层数设置到尽可能的大才能取得最高的加速比。举例而言，图 4.6 展示的是另外一组测试结果。其中给出的依然是在 Last.fm 这个数据集上执行 D=16 即 $S_C = 272$ 情况下的 ALS 算法在不同层数下的加速比。从图中我们可以看到，最高的加速比是在将 L 设置成 16 时取到的（约为 3.8 倍）。这是因为，随着层数的上升，边数据被重复读取的次数也不断增加，这与点权数据重复读取次数的减少构成了一个 tradeoff。而相对于图 4.5，图 4.6 中所展示的情况来自于 D 更小的测试用例，所以其对点权的访问占磁盘总访问量的比值较小。因此，程序员有必要巧妙的设置 L，从而达成最高的加速比。

## 4.5　本章小结

本章描述了一种新型的外存图数据存储格式。其通过将数据图中的点数据分层存储使得一次可以载入到内存中的点数增加，从而减少了点数据被重复载入的次数。作为代价，这一方法会使得边数据被重复载入的次数增加。不过，通过测试发现，只需选取一个合适的层数来达成这一增一减之间的平衡，就能够显著的降低外存图计算系统总体的载入数据量。基于这一发现，我们改进了已有的外存图计算系统 GridGraph，将其扩展成了可以支持分层点数据存储的 3DGridGraph。根据我们的测试结果，3DGridGraph 最高可以比 GridGraph 要快上 4 倍。

# 第 5 章　矩阵计算引擎的自动优化

## 5.1　本章概述

正如前文第 2.3 节中所提到的，由于基于矩阵的执行引擎具有系统 cache 利用率高等特点，其已经被很多追求效率的图计算系统所采用[5,15,54]。事实上，除了图计算系统之外，目前还有越来越多的分布式计算框架也采取以数组为基础的方式进行构建。经测试，这一类的工作普遍比对应的其它类型的系统要快上五到十倍左右[15,32]。然而，虽然已经达成了较高的效率，已有的基于矩阵的执行引擎中的大部分系统都采用了一种最原始的运行方式，因此仍然还有巨大的提升余地。具体来说，这些系统一般来说都将每一个矩阵操作孤立开来单独进行执行。同时，它们的程序的执行顺序也是严格地按照程序员给定的顺序进行的。这就好像是单机程序在编译时完全关掉了编译优化，使用 “-O0” 选项进行编译。虽然这种方法实现起来最为简单，但很明显还存在着极大的优化空间。为了解决这一问题，我们提出了一个基于数组的被称为 Kasen 的新型编程模型。Kasen 通过定义一个严格的计算和通信模型，使得较为精确地对程序进行分析并衡量他们的表现成为可能。基于这样的功能，我们设计了一个 Kasen 的优化器，可以自动对于程序员编写的原始程序进行高级优化。根据我们的评估，Kasen 的优化器可以显着地减少对内存的读/写，缓冲区的内存开销和网络流量。在我们的实验中，其最高可以将原程序加速 5.82 倍。

### 5.1.1　背景介绍

传统的基于数组的编程语言，例如 MATLAB[55] 和 R[56] 语言，能够简洁地表达对于数据的操作，因此它们常常被用于数据分析领域。不过，这一类的编程语言基本都是为单机系统所设计的，没有办法自动地进行横向扩展。随着近期数据量的飞速增长，单机系统的性能往往无法应对海量数据处理的需求。因此，为了在大型数据集上依然可以借用数组型编程接口的表现力，目前许多大数据工具都在上层包装出了了一个类似的矩阵型接口[54]。在这种方案下，这些系统就同时拥有了矩阵型编程语言良好的表达能力和下层大数据处理系统的横向扩展能力。不过，由于这些传统大数据处理框架（如 MapReduce）底层原语的限制，这些工具通常无法做到以最高效的形式执行矩阵运算。因此这一类模拟矩阵接口的系统往往性能并不高[31,57]。因此，目前的一个趋势是直接基于分布式数组操作建立可扩展的计算框架。这些框架直接使用分布式的（N 维）数组来表示数据，并且利用现有的先

进高性能计算技术高效地执行不同的数组操作。举例而言，他们会使用我们前文第 2.3 节中所提到的希尔伯特曲线等技巧优化系统 cache 的利用率。得益于这些技巧的采用，这些基于数组的框架相较于早期的系统往往能够取得显著的性能提升。只不过，虽然这些系统已经比只是模拟矩阵接口的那些系统要快得多了，它们的系统架构中依然缺少了一个重要部分，即优化器。大多数现有的框架都采取了一种较为简单的实现方法，即每个数组操作是单独执行的，并且其执行顺序完全按照程序员定义的顺序执行，这意味着存在巨大的优化空间。举例来说，Spartan[32] 是一个 NumPy[58] 的分布式实现，其中采用了一些简单的优化。根据他们的评测，Spartan 仅仅通过结合连续的两个 Map 操作和连续的 Map-Reduce 操作组的方式就达到了两倍的加速。可以想象，一个支持更复杂优化的优化器可以达到更好的效果。

### 5.1.2　挑战

事实上，与这些分布式的基于矩阵的计算框架不同，在单机矩阵运算程序的领域，通过优化本地数组程序来达到加速效果的方式已经较为成熟[59,60]。有许多的专门针对单机矩阵程序的优化器和优化方法被提出和实现，并取得了很好的效果。但是，由于这些工具都没有考虑到分布式环境下所需要的数据同步等分布式环境下独有的问题，因此它们都不能直接运用于分布式数组程序中。考虑到这种情况，我们希望能够借鉴这些单机的数组程序优化方法设计一个适用于分布式环境的优化方案。这一方案要求不仅能够有效的减少本地的计算时间还要可以减少节点之间的通信成本。

为了达成这一目标，我们借鉴了我们已有的手工优化分布式机器学习程序的经验，总结出三条有效的优化方式：（1）充分利用缓存中的数据，从而减少对于内存的读写；（2）复用缓冲区，从而降低内存的占用和分配内存导致的开销；（3）调整程序的执行顺序，从而降低通信的成本。虽然上述的三种优化方法的思想可以应用于大多数的算法，但是这些优化的具体使用方式高度依赖于算法的实际情况，因此它们通常被认为需要通过程序员手动进行优化。

为了将上诉步骤自动化，我们解决了以下几个难题。首先，根据我们的调查，现有的基于数组的计算框架通常为用户提供了种类过于丰富的算子和使用点对点通信的能力[31,32]。虽然这些特性方便了程序员的使用，但是由于这些框架的自由度太高，其程序行为难以分析，也就很难做出优化。因此，我们提出了一个新的编程模型，其既有足够的限制使得能够进行有效的自动优化，又具有足够强的通用性，可以实现多种算法。其次，基于数组的程序经常会生成大量的临时数组[32]。为

了解决这个问题，Spartan 尝试进行了一些简单的融合优化，比如说将相邻的两个 Map 函数融合成一个 Map 函数以减少一个中间变量等等。这一简单的方法在性能上取得了显著的改善。但是，Spartan 所使用的方法受到原语的类型和输入数量的严格限制，因此通用性不高。为了增强其通用性，我们设计了一个通用的融合优化器，可以对多种操作进行优化。最后，现有的框架同步数据的过程总是要么完全采用立即执行（eager syncronization）的方式或完全采用懒操作（lazy syncronization）的方式执行的。在前一种方案下，数据一旦出现不一致就会立即执行一次系统同步。相对的，在后一种情况下，系统的不一致会一直拖到这一数据需要被输出或者需要被用于某些特殊的操作如选择肢判断等时候才会被同步。不过，经过我们的调研发现。这两个选项都不是最佳的，因此，我们通过组合这两种方法提出了一个新型的通信模式。

### 5.1.3　我们的贡献

总的来说，本章我们将描述了一种新型的基于数组的编程模型 KASEN。类似于 MPI 标准[61]，KASEN 仅仅给出了一系列的程序的标准，没有提供精确的实现。然而，它通过（1）定义三种存储状态；（2）以显式的存储状态转换定义通信；和（3）限定原语的数量等方式对数据的存储和转换进行了严格限制。通过施加这些限制，KASEN 是得一起编写的程序在数据、计算、和通信等三方面都具有极强的可控性和可测性，从而使我们能够设计一个有效的优化框架。这里，“可控性”和“可测性”的意思是我们可以较为精确的分析出可以对程序进行怎样的优化，甚至计算出进行了这样的优化之后能够得到多大的性能提升。同时，由于我们的优化算法与底层各个矩阵操作的具体是解耦的，所以对于任意一种具体的实现方法我们的优化算法都是可行的。根据我们的测试，KASEN 能够用于实现许多重要的机器学习算法和图算法。更重要的是，根据我们对于不同算法的定量分析，KASEN 的优化器可以显著减少程序对内存的读写，缓冲区的分配和通信量。为了验证我们的分析结果，我们还实现了一个简单的 KASEN 原型系统。在此基础上进行的性能评估表明，我们的优化器最高可以将原程序加速 5.82 倍。

## 5.2　设计思路与原理

KASEN 提供了一系列严格约束的原语集合，使得利用 KASEN 写出的程序有更强的可解析性，从而使我们能够进行有效的优化。在这一节中，我们将描述 KASEN 的主要设计思路和相关的基本原理。

1. KASEN 只提供两种高级数据结构即 Vector 和 Matrix。这意味着如果 KASEN

的用户希望将他们的数据以分布式的方式（即每个计算节点仅保存一部分而不是全部的数据）被存储在集群之中，他们就需要将数据事先组织成一维的向量（Vector）或者二维的矩阵（Matrix）的形式。除此之外，其它的数据只能够被定义为 Shared 型共享变量。这意味着数据将会在每个计算节点中都保留一份完整的备份。在对数据的类型进行了限制之后，更进一步的，KASEN 通过仅提供有限的编程原语的方式对可以在这些高阶数据结构上执行的操作种类也进行了限制。具体来说，KASEN 提供了（1）三种基于向量的运算来操作数组，分别是 Map、Reduce、ZipWith；和（2）一个向量与矩阵相互作用的操作 MxV。所有的这些操作的输入和输出都为一整个的 Vector 或者 Matrix 变量。换句话说，KASEN 不允许仅仅对用户的一部分进行访问和操作。这些限制施加复杂优化的前提条件。不过，我们后文的第 5.3.4 节中会提到，虽然施加了这么多的限制，但是 KASEN 依然具有良好的编程能力。许多的重要数据挖掘、机器学习、以及图计算算法都可以用 KASEN 的模型进行表达。

2. KASEN 显示地将节点间通信从本地计算中分离了出来。KASEN 中定义的变量可以存储在三种不同的存储状态中，而且只可以通过进行网络通讯的方式在这些状态之间进行转换。换言之，KASEN 中所有的计算原语都是在本地执行的，只有存储状态的转换进行网络通讯。这种细化的好处有两个：（1）它使得 KASEN 程序中的每一步都是可测量的，这样我们就可以估计每一步的计算开销或者通信成本；（2）它使得程序框架自然符合了 BSP 模型[26]，因此我们可以激进地进行本地的优化而不用担心影响其它计算节点的执行。换句话说，KASEN 的优化器可以在本地执行阶段通过等价转化原始程序的方式来提高性能（例如，通过启发式地在立即同步和懒同步两种方式之间进行切换减少网络开销），而不会影响程序的正确性。再将本地计算和全局通讯划分开来之后，我们仔细研究了这些计算和通讯的原语与存储状态所有可能的组合形式。基于这些规则我们可以对用户初始给出的程序进行等价的变换从而在保证不改变原程序正确性的前提条件下达到最小的通讯量。具体的优化方法我们将在后文的第 5.5 节详细介绍。

3. 通过以上的两个步骤，我们现在可以集中在每个本地计算周期的优化之上，这意味着理论上 KASEN 能够利用本地的数组优化器（如 BTO[59]）进行这一类的优化。不过由于融合的难度，事实上 KASEN 是实现了一个自己的优化器。这一优化器的原理虽然十分简单，但是效果很好，可以在很多实际的情况下达成不错的优化效果。具体来说，我们的优化器基于的是循环融合的思想。我们通过调研发现，在实际的矩阵程序中，有许多的矩阵操作虽然执行的计算不同但是拥有相同的数据访问模式。在这种情况下，这些操作可以被融合成同一个更加通用化的 X-in/Y-out 的通用计算原语，从而通过将这些操作在内部流水线化的方式大大地提

升系统 cache 的利用率。只不过，在分布式的情况下为了保证这一融合的正确性需要对相应的数据进行协同划分，并且执行的计算也需要满足一定的约束限制。举例而言，如果我们的需求是计算两个向量的点乘，在普通的模式下，我们首先需要计算一个 ZipWith 操作从而求得这两个向量对位的乘积。其后，我们还需要执行一个 Reduce 操作对上一步产生的中间结果进行累和，从而得到最终的结果。通过将这两个操作融合成一个，我们可以以流水线的形式将 ZipWith 的输出在还没有被逐出 cache 的情况下直接给到 Reduce 操作进行累和，从而节省掉对中间结果对内存的写入和读取。根据我们的测试结果，这一简单的优化可以在很多情况下都达成很好的效果。

总体上来说，由于 KASEN 拥有严格的编程模型，用它可以很容易地将程序的数据依赖关系转化成一个表示依赖关系的数据依赖有向无环图（DDAG）。对于一个 DDAG，每个节点代表一个数据变量，每条边代表一个数据的依赖关系。DDAG 图的边分为两种：（1）本地计算的边代表了本地计算原语；（2）全局通信的边代表着存储状态的转化。这些全局通信的边将整个 DDAG 转化为若干个 DDAG 子图，而每一个 DDAG 子图都可以像本地数组程序一样进行优化。因此，一个简单的基于 KASEN 模型的优化器就可以通过枚举所有 DDAG 的等价转化形式，再优化各个 DDAG 子图的方式求得最优的结果。不过，这样的优化方式的复杂度是随着 DDAG 中点的数量指数级增长的，因此不适合于大的程序。为了解决这一问题，我们将在第 5.5 节中描述一个基于贪心的近似求解方法。

## 5.3　KASEN 的编程模型

利用 KASEN 模型进行编程是一件非常方便的事情。其过程仅包括如下的两个步骤：(1) 利用 KASEN 提供的原语就像在单机上编程一样进行编程；和（2）将组织成向量和矩阵的数据均等地划分给不同的计算节点。除此之外所有的细节，比如说如何对数据进行同步，如何进行存储状态的转换等等都会由系统自动完成，不需要程序员去干涉。为了更进一步地说明这一过程，在本节中，我们将首先描述如何在 KASEN 模型下定义数据。然后我们会分别介绍 KASEN 提供的几个操作接口，并用 PageRank 作为例子举例说明在 KASEN 模型下的编程方式。

### 5.3.1　数据

**数据的定义**　正如我们上文提到的，KASEN 只提供两种高级数据结构即 Vector 和 Matrix。使用这两种类型定义的变量能够被分割存储到不同的计算节点之中。除此之外，数据还能够被定义为 Shared 型共享变量，这意味着数据将会在每个节点中

都保留一份完整的备份。具体来说，变量将通过以下形式定义：

**Shared:** Shared<C-style type>

**Vector:** Vector<C-style type>[Length]

**Matrix:** Matrix<C-style type>[Height, Width]

这些定义的格式非常类似于 C++ 语言 STL 库中定义的那些容器。它们实质上也是很相似的，都是表示了一组用户的变量，并限制了可以对这一组变量进行的操作。除上述的简单定义方法之外，用户还可以用 C 语言的文法定义固定大小的 Vector 或者 Matrix 数组，比如说 *"Vector<int> tmp[10]"*。而且，Matrix 变量的元素类型还可以指定为 null，这意味着这是一个 0/1 矩阵，没有附带的元素需要存储。

**数据的划分**　KASEN 中定义的 Vector/Matrix 变量将会被均分到所有的计算节点之中，其中每个计算节点都会被分到"独有"的不相交的一部分。另外，为了能够做到计算原语仅仅进行本地计算，我们对于数据大小相同的 Vector 或者 Matrix 变量将会进行协同地数据分割。具体来说，如果向量 V1 和 V2 具有相同的长度，那么 V1[i] 和 V2[i] 将会被分配到同一个计算节点之上。同样，如果矩阵 M1 和 M2 具有相同的高度和宽度相同，M1[i][j] 和 M2[i][j] 也会被分配到相同的计算节点之上。虽然添加了这样的限制，但是正如我们在前文第 5.1.3 节中提到的，KASEN 的优化算法是与下层的具体实现节藕的。因此在后文计算通讯量的减少时单位全部都是百分比的形式。无论用户选用了何种的数据划分方法这样比例的通讯量减少都是可以达到的。只不过，为了方便起见我们推荐用户使用 hybrid-cut 作为稀疏矩阵的划分方法，以及用基于数据范围的划分方法去划分稠密的向量和矩阵。

### 5.3.2　数据的操作

KASEN 对能够在上述高级数据结构上进行什么样的操作有着严格的限制。具体来说，KASEN 提供的原语可以分为以下三类：

**共享的运算操作**　由于每个节点都拥有共享变量的一个备份，程序员能够编写任意函数操作共享变量。换句话说，每一次对共享变量的操作实际上会被翻译成每一个计算节点对本地缓存的这一变量的副本各自进行的一次操作。由于这些操作都是确定性的，所以事后的结果也都是一样的，没有进行同步的需要。

$$Shared\ \ ResS = F(Shared\ \ Arg1,\ Shared\ \ Arg2,\ \ldots)$$

**只支持数组的运算操作**　不同于共享操作，KASEN 定义了三种基于向量的运算来操作数组，其中 Map 操作的输入和输出都是一个类型为 Vector 的向量。其具体过程就是将用户给出的自定义函数在输入的向量中的每一个元素上操作一下，从而得到输出的向量中的元素。

$$
\begin{aligned}
&Vector\ \ ResV = Map(Function\ \ F,\ Vector\ \ V,\ Shared\ \ Arg)\\
&\quad in\ \ which,\ \ ResV[i] = F(V[i],\ Arg)
\end{aligned}
$$

相对的，Reduce 函数的输入虽然也是一个类型为 Vector 的向量，但是它的输出仅为一个类型为 Shared 的变量。用户给定的二元函数 F 被用于将这一输入向量中的所有元素综合成一个 Shared 变量。在这里我们要求用户给定的这个二元函数同时满足交换律和结合律，因此事实上这个输入向量 V 的元素是以何种顺序被迭代的都不影响输出的结果。

$$
\begin{aligned}
&Shared\ \ ResS = Reduce(Function\ \ F,\ Vector\ \ V)\\
&\quad in\ \ which,\ \ ResS = F(V[0], F(V[1], F(V[2], \ldots)))
\end{aligned}
$$

最后一个操作是 ZipWith，其过程与 Map 操作类似但是要求输入的有两个类型为 Vector 的向量，输出为一个 Vector 向量。具体的过程则是将用户给定的函数作用于输入的两个向量中对位的元素，从而得到输出向量中的元素。和之前的两个操作不同，ZipWith 是一个二入一出的操作，因此可以和其他操作配合用于组装更复杂的操作。

$$
\begin{aligned}
&Vector\ \ ResV = ZipWith(Function\ \ F,\ Vector\ \ V1,\\
&\qquad\qquad\qquad\qquad Vector\ \ V2,\ Shared\ \ Arg)\\
&\quad in\ \ which,\ \ ResV[i] = F(V1[i],\ V2[i],\ Arg)
\end{aligned}
$$

注意上述的三种基于向量的运算拥有相同的内存访问模式。它们都是通过在输入向量中的元素上进行迭代得出的输出结果，而且都不关心这一迭代的进行顺

序。这一特点使得它们可以通过流水线的方式融合成一个操作进行计算，唯一的要求仅仅是输入的向量们是协同划分的。

**矩阵和向量间的运算操作**　除了单纯针对于向量的操作，KASEN 还定义了一个稀疏矩阵和向量间的操作 MxV，定义形式是：

$$
\begin{aligned}
Vector\ \ ResV = MxV(&Function\ \ F_Z,\ Function\ \ F_R,\\
&Vector\ \ V,\ Matrix\ \ M)\\
in\ \ which,\ \ ResV[i] = &Reduce(F_R, ZipWith(F_Z, V, M[:,i]^T))
\end{aligned}
$$

这一操作的输入为一个长度为 H 的向量和一个大小为 H 行 W 列的矩阵，输出为一个长度为 W 的向量。其计算的数据访问模式基本上和一个普通的向量矩阵乘法是相同的，只不过各个基本计算操作被替换成了用户自定义的函数。具体来说，输出向量中的第 i 个元素的结果就来自于（1）将输入向量 V 和输入矩阵 M 的第 i 列用用户函数 $F_Z$ 进行一次 ZipWith 操作；（2）再将上述操作产生的中间结果用用户给定的函数 $F_R$ 执行一次 Reduce 操作从而得到最终的一个 Shared 元素。和普通的 Reduce 计算相同，为了方便在计算时只执行局部计算，我们要求 MxV 操作中的用户指定函数 $F_R$ 也是同时满足交换律和结合律的。正如我们后文第 5.5 节中会讨论到的，多个连续且共享同一个输入矩阵的 MxV 操作也可以融合成一个矩阵与矩阵的乘法进行计算，从而提升系统 cache 的利用率。除此之外，我们还定义了操作 MxVT，它的参数类型和 MxV 是相同的，但是使用了转置矩阵作为输入。即 MxVT 操作的输入为一个长度为 W 的向量和一个大小为 H 行 W 列的矩阵，输出为一个长度为 H 的向量。其计算模式基本上和将输入向量和输入矩阵的转置 $M^T$ 进行矩阵向量乘法类似。通过上述的叙述我们可以看到，在 KASEN 模型中我们事实上并不区分稀疏矩阵和稠密矩阵。这是因为我们的 KASEN 模型和具体的实现是解耦的。所以用和可以根据他们的数据类型执行选择最佳的执行方式，而同时我们的优化算法都可以产生作用。

### 5.3.3　实例：PageRank

是在这一小节中，我们将使用 PageRank 算法作为例子更形象的介绍如何使用 KASEN 模型进行编程。PagraRank 问题的数学定义可以被看作是为了求解出一张图上每一个点的 “page rank’’ 值，其用途非常广泛，在网页排序等重要例子中都有

**算法 4** 基于 KASEN 模型实现的 PageRank 算法。

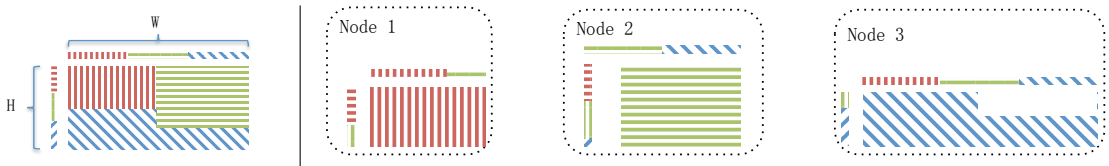
**Data:**
Shared<int> N = load();
Matrix<null>[N, N] G = load();
Vector<double>[N] pr = {1}, new_pr, tmp1, tmp2, degree;
Shared<double> diff, r = load(), theta = load();
Shared<bool> not-converged = True;
**Definition:**
$F_1(x,y) = x/y$
$F_2(x,r) = (1-r) + r*x$
$F_3(x,y) = abs(x-y)$
$F_4(x) = x$
**Computation:**
1: degree = **MxVT**(*, +, G, pr);
2: **while** not-converged **do**
3: tmp1 = **ZipWith**($F_1$, pr, degree);
4: tmp2 = **MxV**(*, +, G, tmp1);
5: new_pr = **Map**($F_2$, tmp2, r);
6: diff = **Reduce**(+, **ZipWith**($F_3$, pr, new_pr));
7: pr = **Map**($F_4$, new_pr);
8: not-converged = (diff > theta);
9: **end while**

应用。这一算法的基本实现是首先随机初始化所有点的初始 “page rank’’ 为 1 然后再迭代的进行更新。在每一轮迭代的过程中，每一个点的 page rank 值是按照 $pr^{t+1}[v] := (1-r) + r*\sum_{(u,v)\in E}(pr^t[u]/degree(u))$ 的公式进行的。在这一公式中 r 是控制随机跳跃频度的系数，E 是数据图 G 的边的集合，$pr^t[u]$ 表示图中点 u 在第 t 轮迭代结束后的 page rank 值，degree(u) 则是点 u 的出度，即出边个数。

算法 4 给出了基于 KASEN 模型实现的 PageRank 算法。具体来说，这一程序首先使用了一个 MxVT 操作计算出了每一个点的出度。其后 while 循环中的每一轮迭代都是上述公式的直接翻译，只不过在最后添加了一段判断是否收敛以便结束循环的代码。从中我们可以看到，使用 KASEN 模型进行编程时事实上并没有在分布式环境下还是在单机环境下编程的区别。

### 5.3.4　应用范围

虽然 KASEN 目前只支持有限制的原语，但是它已经能够高效的实现许多机器学习算法和图算法。举例来说，L-BFGS 算法[62] 可以由 KASEN 实现，而 L-BFG 算法又可以用来解决许多本质上等同于求极值的问题，如线性回归、支持向量机（SVM）等。此外，KASEN 框架通过 MxV 和 MxVT 运算相结合能够完全等效于使用 PowerGraph 的 GAS 模型进行编程，因此 PowerGraph[3] 支持的所有的图算法也都可以通过 KASEN 来实现。




(a) Partitioned 状态　　(b) Consistent 和 Pre-reduced 状态

图 5.1　三种不同的数据存储状态。

## 5.4　KASEN 模型的具体实现

上一节中我们仅仅是从一个高层次的角度讨论了一个基于 KASEN 模型的程序是怎么编写的，并没有涉及这一程序应该具体的怎么被下层的执行引擎所执行。因此，我们将在这一节中详细描述 KASEN 模型对执行引擎的具体要求，包括（1）用户定义的数据是如何分布式的存储在集群中的；（2）这些数据在操作的过程中变的不一致了之后是如何同步的；以及（3）特定的操作原语在操作特定的数据时有哪些关于数据存储状态的约束。

### 5.4.1　三种数据存储状态

正如前文第 5.3.1 中所说的，KASEN 要求用户为每一个 Vector 和 Matrix 类型的变量给出一个划分的方法。KASEN 本身对这一划分仅仅作出了一些比较简单的限制：（1）数据需要被尽可能均等地将数据划分成和集群中计算节点个数相等份的不相交的子集；（2）长度相同的向量或者大小相同的矩阵需要被给予同样的划分方法。在这种情况下，如果分属两个长度相同的向量或者大小相同的矩阵的两个元素具有相同的下标，他们需要被分配给同样的一个计算节点。在后续的论文中，我们将把一个被存储在其被分配的计算节点的元素称之为 master。对于一个 Vector 或者 Matrix 类型的变量来说，如果每一个计算节点仅仅存储被指派给它的元素，那么我们将这样一种状态称之为 Partitioned 状态。如果一个 Vector 或者 Matrix 类型的变量被存储在 Partitioned 状态下，我们通过添加一个 P 的下标来表示它，比如说 $V_P$ 和 $M_P$。

不过，由于向量和矩阵元素的划分是分开进行的，所以在进行 MxV(T) 计算的时候有可能会需要一些并没有被分配给特定计算节点的数据。具体来说，如果一个下标为 (i,j) 的矩阵元素被分配给了计算节点 k，那么为了计算 MxV 操作，对应的输入向量的第 i 个元素和对应的输出向量的第 j 个元素也需要在这个计算节点上存在一个副本才行。为此，除了上述的 Partitioned 状态之外我们还定义了一种被称之为 Consistent 的状态。在这种状态下每一个计算节点除了维护被分配给它的 master 元素之外还会存储一些 slave 元素，它们值和对应的 master 相同。具体来说，如果一个大小为 H 行 W 列的矩阵的一个下标为 (i,j) 的非零元被分配给了某

一个特定的计算节点，那么每一个长度为 H 的向量的元素的第 i 个元素和每一个长度为 W 的向量的第 j 个元素要么直接被指派给了这个计算节点，要么这个计算节点会维护一个这个元素的 slave。在这篇论文中，我们将一个被存储在 Consistent 状态下的 Vector 变量用添加下标 C 的方法表示，即 $V_C$。

除上述两种状态之外，我们还允许 Vector 类型的数据变量被存储在一种被称之为 Pre-reduced 的状态下，用于表示在进行了某些操作之后的中间状态。对于每一个被存储在 Pre-reduced 状态下的 Vector 都需要附带一个附带的归一函数 R，这一函数也需要同时满足交换律和结合律。在这种情况下，一个 Vector 变量世纪的存储方式基本上和 Consistent 状态下是一样的，也分为 master 和 slave。只不过 pre-reduced 状态下的 slave 的值并不是相应 master 的拷贝。相反，每一个 master 和其对应的所有的 slave 的值组合起来才能够得到最终的结果。具体的组合方式就是使用归一函数 R 进行 Reduce 操作。举例来说，如果一个元素的 master 值为 a，其含有两个 slave，值分别为 b 和 c，那么这一个元素的实际值就是 R(a,R(b,c))。再举另外一个例子的话，如果们尝试在各个计算节点的本地计算一个处于 Consistent 状态下的向量 $V_C$ 和一个处于 Partitioned 状态下的矩阵 $M_P$ 的乘积的话，由于 Consistent 状态已经把所有需要的输入数据准备好了，所以计算的过程中并不需要进行任何附加的通讯操作。而这一操作的结果就是一个存储在 pre-reduced 状态下的输出向量，其归一函数 R 为累和函数 + 。这意味着最终的结果实际上是将每一个元素的 master 和其对应的所有 slave 累和起来得到的。这一 pre-reduced 状态的存在使得我们可以推迟某些特殊操作所需的通讯，从而给了我们更大的优化空间。与前面两种状态相似，为了表示一个 Vector 类型的变量是被存储在 pre-reduced 状态下的我们会在相应变量的名称上加一个 PR(R) 的下标，即 $V_{PR(R)}$。其中 R 为对应的归一函数。

图 5.1 给出了各种情况下的数据的一个图示。其中子图（a）给出的是用户对这一数据的划分，被分配给不同计算节点的元素用不同的颜色进行了标记。存储在这种情况下的时候数据变量被称之为被存储在 partitioned 状态下。相对的，子图（b）中给出了在 consistent 状态和 pre-reduced 状态下实际会被存储的数据。从中我们可以看到，每一个计算节点不仅仅存储了被分配给它的 master 元素，还存储了根据分配给它的 Matrix 类型变量对应的 slave 元素。这两种状态在存储的元素上是一样的，唯一的区别在于（1）在 consistent 状态下，所有的 slave 的值和对应的 master 是相同的；而（2）在 pre-reduced 状态下元素实际的值是 master 和所有的 slave 被用归一函数 R 综合起来后的结果。

和 Vector 类型的变量类似，Shared 类型的变量也可以被存储在 Partitioned、

Consistent、和 Pre-reduced 三种状态下，分别表示为 $S_P$，$S_C$，和 $S_{PR(R)}$。其中，在 partitioned 状态下 Shared 变量的值仅仅被存储在对应的 master 上，consistent 状态下每一个计算节点都会持有相等的一份数据，最后在 pre-reduced 状态下需要使用对应的归一函数 R 将所有的值综合起来才能得到实际的值。

### 5.4.2　显示的存储状态转化

上述三种存储状态的显示转化是单向的。即只能从 pre-reduced 状态转化成 partitioned 状态，然后再从 partitioned 状态转化到 consistent 状态。根据定义，一个存储在 pre-reduced 状态下的 Vector 类型变量 $V_{PR(R)}$ 可以借由一个 gather 操作转化成 partitioned 状态。具体来说这个 Vector 中每一个元素被指派的 master 会负责将相应的 slave 的值收集过来，利用归一函数 R 进行综合从而计算得到最终的值。在这之后，一个 partitioned 状态下的 Vector 类型变量 $V_P$，可以借由一个 scatter 操作转化成 consistent 状态。具体的过程是向量中的每一个元素的 master 将这一元素的值发送给对应的 slave。与 Vector 类型的变量类似，Shared 类型的变量也可以通过 gather 和 scatter 操作从 pre-reduced 状态转化成 partitioned 状态，再从 partitioned 状态转化到 consistent 状态。与之相对的，KASEN 模型中定义的 Matrix 类型的变量只能被存储在 partitioned 状态下。读者可能会发现，gather 和 scatter 这两个操作时所需要的通讯量和实际的数据划分方法是有关的。不过，正如我们在后文第 5.8 节中会描述的，我们的优化算法所做的是削减需要的 gather 和 scatter 操作的数量，因此我们的优化算法的实际效果和实际的实现方法与划分过程是解耦的。这意味着无论用户给定的是何种的划分，我们的优化算法能够优化的比例是相同的。

### 5.4.3　约束条件

由于我们的 KASEN 模型限定每一个计算操作的执行仅仅会进行本地的计算而不会进行任何远程的通讯，因此我们需要对什么样的操作可以在什么样的数据存储状态下执行作出限制。具体来说，我们将所有合理的组合条件列在表 5.1 中。其中的模式可以为总结如下的几个规则：

1. 所有的 Shared 类型的变量在执行操作之前都必须要先行转化成 consistent 状态。这是因为我们允许对 Shared 类型的变量进行任意种类的操作。

2. 相对的，Reduced 操作的输入 Vector 可以被存储在 partitioned 状态下。除此之外，这一输入向量还可以被存储在 pre-reduced 状态下，只不过在这种情况下用户给定的 Reduce 函数 R 需要和这一 pre-reduced 状态下的向量的归一函数 R 是一样的。Reduced 函数的输出变量只能是一个存储在 pre-reduced 状态下的 Shared 类

表 5.1　所有合法的计算原语和存储状态的组合。公式 (11) 和 (15) 上标注了一个 $\star$。这意味着在这两种情况下，操作仅仅在 master 元素上执行。

| **Explicit storage-state transition** | |
|---|---|
| (1) $V_{PR(R)} \xrightarrow{\text{gather}} V_P \xrightarrow{\text{scatter}} V_C$ | (2) $S_{PR(R)} \xrightarrow{\text{gather}} S_P \xrightarrow{\text{scatter}} S_C$ |
| **Operating shared variables** | |
| (3) $ResS_C = F(Arg1_C,\ Arg2_C,\ \ldots)$ | |
| **Reduce** | |
| (4) $ResS_{PR(F)} = Reduce(F,\ V_P)$ | (5) $ResS_{PR(F)} = Reduce(F,\ V_{PR(F)})$ |
| **MxV(T)** | |
| (6) $ResV_{PR(F_R)} = MxV(F_Z,\ F_R,\ V_C,\ M_P)$ | (7) $ResV_{PR(F_R)} = MxVT(F_Z,\ F_R,\ V_C,\ M_P)$ |
| **Map** | |
| (8) $ResV_P = Map(F,\ V_P,\ Arg_C)$ | (9) $ResV_C = Map(F,\ V_C,\ Arg_C)$ |
| (10) $ResV_{PR(R)} = Map(F,\ V_{PR(R)},\ Arg_C)$ *iff* $F(R(a,\ b),\ Arg) = R(F(a,\ Arg), F(b,\ Arg))$ | |
| (11) $ResV_{PR(R)} = Map(F,\ V_{PR(R)},\ Arg_C)$ *iff* $F(R(a,\ b),\ Arg) = R(F(a,\ Arg),\ b) = R(a,\ F(b,\ Arg))^\star$ | |
| **ZipWith** | |
| (12) $ResV_P = ZipWith(F,\ V1_P,\ V2_P,\ Arg_C)$ | (13) $ResV_C = ZipWith(F,\ V1_C,\ V2_C,\ Arg_C)$ |
| (14) $ResV_{PR(R)} = ZipWith(F,\ V1_{PR(R)},\ V2_C,\ Arg_C)$ *iff* $F(R(a,\ b),\ c,\ Arg) = R(F(a,\ c,\ Arg), F(b,\ c,\ Arg))$ | |
| (15) $ResV_{PR(R)} = ZipWith(F,\ V1_{PR(R)},\ V2_P,\ Arg_C)$ *iff* $F(R(a,\ b),\ c,\ Arg) = R(F(a,\ c,\ Arg),\ b) = R(a,\ F(b,\ c,\ Arg))^\star$ | |

型变量，其归一函数就是用户给定的 Reduce 函数 R。

3. 在执行 MxV 和 MxVT 函数之前，其输入的 Vector 类型变量 V 必须被存储在 consistent 状态下。而这一 MxV 或 MxVT 函数的结果虽然同样是一个 Vector 类型的变量，但是是存储在 pre-reduced 状态下的。相应的归一函数应该被设定为用户指定的 $F_R$ 函数。

4. Map 操作的输入向量同时可以被存储在 partitioned，consisten，和 pre-reduced 这三种状态的任何一种。而其输出向量的存储状态则与输入向量保持一致。只不过，在输入向量的存储状态为 pre-reduced 状态的情况下，这一向量的归一函数 R 和用户给出的 Map 函数 F 需要满足表 5.1 中公式（10）或者公式（11）所描述的约束条件。在余下的论文之中，我们将把中公式（10）所描述的约束条件称之为 exchange law。为了证明这一 exchange law 是能够使得我们的程序运行正确的充分条件，我们需要证明在将用户给定的 Map 函数作用于每一个 pre-reduced 状态下的向量元素之后，其结果会是一个同样处于 pre-reduced 状态下的向量。这一证明可以等价于证明 exchange law 可以被推广到一种更加通用的模式，即 $F(R(a,R(b,R(c,\ldots)))) = R(F(a),R(F(b),R(F(C),\ldots)))$。而这一步的证明非常简单，仅仅需要将原来的 exchange law 连续使用就可以了。

相对的，我们将另外一个公式（11）所描述的约束条件称之为 pushdown law。

因为这一公式的应用就是将一个外层的操作下推给任意一个内层的变量。在这一种情况下，由于实际上每一个向量的元素仅仅对其 master 进行了一次操作，所以其结果的正确性是显然的。举例而言，如果用户给出的 Map 函数 F 是 “$F(x) = x + 1$”，那么如果存储在 pre-reduced 状态下的输入向量的归一函数 R 为累和函数的话这两个函数就是满足 pushdown law 的。在这种情况下，每一个计算节点可以简单的在本地将其所管理的所有 master 加一，不需要进行任何通讯操作，其结果就是一个存储在 pre − reduced 状态下，归一函数为累和的向量。

5. 对于 ZipWith 函数来说，其两个输入向量 V1 和 V2 可以（1）同时存储在 partitioned 状态下；（2）同时存储在 consistent 状态下；（3）一个存储在 consistent 状态下另一个存储在 pre-reduced 状态下。只不过这一情况下，用户给出的 ZipWith 函数 F 和 pre-reduced 向量的归一函数需要满足表 5.1 中公式（14）所描述的约束条件，即满足分配律；或者（4）一个存储在 paritioned 状态下另一个存储在 pre-reduced 状态下。同样的，在这一情况下，用户给出的 ZipWith 函数 F 和 pre-reduced 向量的归一函数需要满足表 5.1 中公式（15）所描述的约束条件，即另外一种形式的 pushdown law。拿最常见的一种归一函数 + 来说的话，如果用户给定的 ZipWith 函数 F 是乘积函数 * 的话，这两者就是满足分配律的。而如果如果用户给定的 ZipWith 函数 F 是减法函数 − 的话，这两者就是满足 pushdown law 的。由于对公式（14）和公式（15）的充分性的证明基本上和对公式（10）和公式（11）的充分性的证明基本是一样的，这里我们不再赘述。

在实际运行的过程中，如果执行 KASEN 程序的执行引擎发现当前的条件不满足表 5.1 中的任意一种形式。那么引擎就需要自动插入相应的 gather 和 scatter 操作对输入向量的存储状态继续转换。我们在下一节中将描述一种可以尽可能降低通讯量的插入方法。

## 5.5　优化方法

为了能够进行尽可能有效的优化，我们需要尽可能地对可以进行的计算类型进行约束。这也就是为什么我们的 KASEN 模型给出了出仅仅有四个的操作类型。不过虽然仅仅提供了四个操作，我们的模型的编程能力并没有被限制，同样可以用于实现很多重要的算法。更进一步的，这四个操作中（1）三个以向量为中心的操作 Map、Reduce、ZipWith 都可以很简单的用一个循环操作实现，其效率也非常的高；同时（2）另外一个向量与矩阵的乘积操作 MxV 也是被很多现成的矩阵运算库提供了的[5,63]。然而，虽然在单独执行的情况下已有的方法可以很简单地将这些操作的执行以硬件能够提供的最高的速度执行，但仅仅这样是不够的。为了能

够得到最佳的效率，我们还需要进行能够跨越操作边界的优化。

### 5.5.1　循环融合

数据局部性的好坏对于矩阵类程序的性能起着非常重要的作用。特别是对于所谓的大数据应用来说，程序数据局部性的好坏所起到的作用更加明显。正如大数据程序这一程序类别的名字所描述的，这一类的程序所处理的一般都是规模十分庞大的数据集。在这种情况下，为了保证其计算的时间依然是可以接受的，一般的大数据应用的复杂度都不高。一般情况下其复杂度都不超过所谓的近似线性（quasilinear）地复杂度，即 $O(n*log^k n)$。这一复杂度意味着这些程序基本上都是内存带宽敏感（bandwidth sensitive）的程序[59]，其运行速度基本上与能够从内存将数据载入到 cache 中的速度成正比。在这种情况下，增加程序数据的局部性，从而提升系统 cache 的利用率的效果就非常的明显。

```
for i = 1:N
   new_pr[i] = F2(tmp2[i], r)
for i = 1:N
   _tmp[i] = F3(pr[i], new_pr[i])
diff = 0
for i = 1:N
   diff = diff + _tmp[t]
for i = 1:N
   pr[i] = F4(new_pr[i])
for i = 1:n
   tmp1[i] = F1(pr[i], degree[i])
```

(a) Before fusing

⇒

```
diff = 0
for i = 1:N
   new_pr = F2(tmp2[i], r)
   _tmp = F3(pr[i], new_pr)
   diff = diff + _tmp
   pr[i] = F4(new_pr)
   tmp1[i] = F1(pr[i], degree[i])
```

(b) After fusing

图 5.2　循环融合.

为了达成提升程序数据局部性的目的，一类被称之为循环融合的技巧在高性能计算领域内被广泛的运用。这一循环融合的技巧主要的目的是将多个连续的循环融合成同一个循环。如果这些原本分开的循环读取了相同的数据或者可以形成某种流水线的形式的话，这样的循环融合就可以极大地提升数据的局部性。拿前文算法 4 中描述的 PageRank 的程序作为例子的话，其中包含在第五到七行和循环的下一轮迭代所要执行的第三行中所包含的五个不同的向量操作就可以通过循环融合融合成同一个循环之内，从而极大地提升数据的局部性。由于对每一个不同的下标 i 来说，完成其相关的所有计算仅仅需要访问九个不同的变量，因此可以完全的被系统 cache 所包含。在这种情况下，融合后的程序（如图 5.2 所展示）仅仅需要读取三个不同的输入向量（即 tmp1, pr, degree）各一次，并写入两个不同的输出向量（即 pr，tmp1）各一次。这一融合一共减少了大约 55% 的内存读写。根据

近期的调研[59]，通过循环融合可以在很多内存敏感的应用的情况下取得非常好的效果。

在 KASEN 模型下，一个基于 KASEN 给出的 API 组合写出的程序同样可以用类似的技巧进行优化。正如我们在前文第 5.3.2 节中所提到的，KASEN 模型提供的所有四个不同的操作可以按照它们的内存访问模式分为不同的两个组合。一组是仅仅访问 Vector 类型变量的 Map、Reduce、ZipWith 这三个操作。另外一组是会同时访问 Vector 类型变量和 Matrix 类型变量的 MxV 和 MxVT 操作。而连续的几个操作如果属于同一个组合的话就可以将用于实现它们的循环融合成一个，从而提升系统 cache 的利用率。不过，将这一技巧运用到 KASEN 模型所编写的程序中并不是一件非常简单的事情。如果直接将所有连续且属于同一组合的操作直接融合起来的话就可能导致违反表 5.1 中所列出的组合的情况，从而导致程序的运行出错。在实际的优化过程中，操作的融合很可能会被上述的约束条件不满足所打断，因此需要插入合适的通讯原语。同时，为了降低总体的通讯量，还需要设计一种合适的方法使得插入的同步原语数量尽可能的少。

在本节的后续论述中，我们会首先描述两种内部操作的执行过程。这两种内部操作分别代表了一种 KASEN 操作的内存访问模式。所有的使用这一内存访问模式的 KASEN API 在实际执行的时候都可以被这一内部操作所替代。在这之后，我们会描述一种叫做 DDAG 的数据结构。借助它我们可以很方便的表示和分析用 KASEN 所编写的程序，从而进行进一步的优化。最后，我们会描述 KASEN 所使用的优化算法。

### 5.5.2　内部操作

为了能够实现上述的循环融合技巧，一种最简单的方式是在 KASEN 已有的操作的基础上再提供一些列的组合操作。比如说，我么可以提供一个拥有三个输入向量一个输出向量的操作 ZipWith3(a,b,c)，用来替代原来需要使用两个 ZipWith 操作实现的 ZipWith(ZipWith(a, b), c)。在这种情况下我们可以仅用一个循环替代原来需要两个循环实现的操作，从而提升程序的数据局部性。不过，这样的方法虽然简单但难以实用化。首先很难能够列出所有需要提供的额外操作组合。即便能够做到这一点，让用户去手动使用这大量的 API 也非常的繁琐。

相对的，KASEN 实际采用的方法是为每一种独特的内存访问模式提供一个内部操作。由于 KASEN 的所有操作只包含两种不同的内存访问模式，所以我们仅仅需要实现两种不同的内部操作。同时，用户也不需要直接使用这些内部操作。KASEN 的优化器会负责自动地将贡用户使用的外部 API 自动的转化成内部操作用

于执行。具体来说，KASEN 的内部操作包含如下两种：

**IterOnVec**　正如我们在前文第 5.3.2 节中提到的，所有的三种向量操作 Map、Reduce、ZipWith 都使用的是同一种内存的访问模式。他们都可以用同样的一种方法，即按某种任意的顺序迭代输入向量的所有下标的方式实现。这也就意味着，如果两需的进行很多个向量操作的执行，事实上我们并不需要将他们分开在不同的循环中实现，仅仅需要同一个循环就可以了。即使是在分布式的执行环境下，这样的循环融合依然是可能的。唯二的要求仅仅是用户需要按照我们在前文第 5.3.1 节中所描述的那样对数据进行了系统划分就可以了。在这种情况下，所需的数据全部都已经妥当的被存储在本地节点了，并不需要进行其他的通讯操作。

为了表示这一类的操作，我们定义了一种新的内部操作 IterOnVec 来实际执行这一类的操作。具体来说，IterOnVec 是一个可变参数的操作，其接收一个 X 入 Y 出的用户函数，同时读取 X 个不同的 Vector 或者 Shared 类型的变量，并同时更新 Y 个不同的 Vector 或者 Shared 类型的变量的值。如果在融合前这几个向量操作读取了同样的向量数据，那么这一份冗余的读取就可以被省略掉。同时，如果原来的向量操作中有中间变量的产生的话，通过融合也可以省略掉。

**MxM(T)**　和 IterOnVec 操作类似，MxM(T) 操作也是一个可以用来替代连续执行的 MxV(T) 操作的一个通用变长操作。不过在连续执行这一个条件之外，它还要求这几个 MxV(T) 操作读取的是同样的一个 Matrix 类型变量，且没有组成任何的数据依赖关系。在这种情况下，通过将这几个操作融合成一个操作执行可以节省掉对输入的 Matrix 类型变量的重复读取。就实现来说，MxM(T) 操作的实现基本上等价于一个叫做 “GEneral Matrix to Matrix Multiplication (GEMM)” 的通用矩阵操作。这一 GEMM 操作所需要输入的两个矩阵中一个是原本的输入矩阵，另外一个则是将几个 MxV(T) 操作的输入向量拼接起来所产生的矩阵。由于 GEMM 是一个非常常见且被很多种已有的矩阵库都实现提供了的操作，因此对于其具体的实现我们就不在此赘述了。

### 5.5.3　DDAG

为了方便将 KASEN 提供给外部用户进行使用的操作自动地翻译成内部操作的这一过程，我们提出了一种被称之为 Dependency DAG （DDAG）的数据结构。每一个 DDAG 都可以和一个 KASEN 程序的片段等价的转换，同时也非常的方便供人识别与程序分析。具体来说，DDAG 中的每一个点都代表 KASEN 程序的一个变量，而每一条边都代表一个 KASEN 操作所带来的数据依赖。由于 KASEN 将

所有的操作分为两种不同的操作，因此在 DDAG 中也有两种不同类型的边。一种是所谓的局部计算类依赖，代表一个 KASEN 的计算操作。另一种是所谓的通讯依赖，其代表的是 gather 或 scatter 这两个用于进行数据存储状态转化的操作中的一个。除此之外，我么在每一个变量的名称上都增加了一个下标，用于表示其存储状态这一可以用来判断是否符合约束跳的重要的状态。

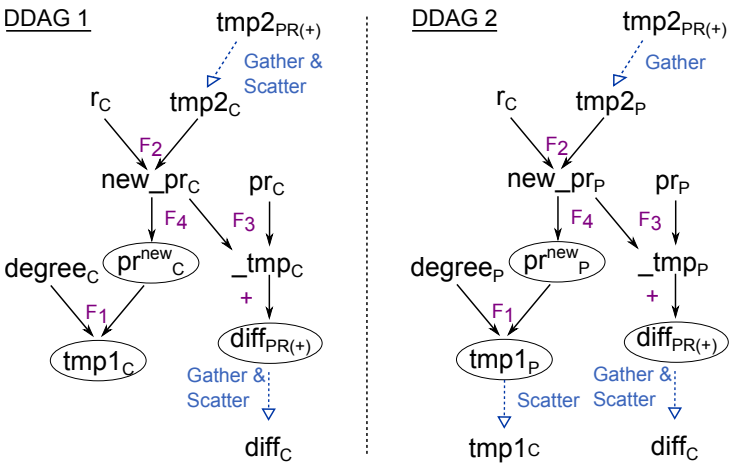



图 5.3　两个等价的 DDAG。

图 5.3 展示了两个不同但是等价的 DDAG，它们都是从前文算法 4 中的第五到七行以及下一迭代的第三行所包含的五个操作转化过来的。在这张图中，每一个变量的存储状态被标记在相应变量名称的下标上。同时，两种不同的边分别用两种不同的箭头所表示。纯黑色的箭头表示本地的计算，而蓝色虚线组成的箭头表示通讯操作。从图中我们可以看到，这两个等价的 DDAG 在黑色箭头所代表的计算部分是相同的，唯一的区别在于使用了不同组合的通讯操作，随之带来的是变量下标所代表的存储状态也不同。与使用 eager synchronization 将所有的通讯都在最上方执行的第一种 DDAG 不同，第二种 DDAG 将一个 scatter 操作从第一个 DDAG 的最上方推迟到了最下方去执行。通过后文第 5.6 节中给出的计算公式我们可以计算得出，虽然这两种情况下所需要的通选量都是一样的，但是第二种 DDAG 所代表的那种执行方式所需要的计算量较少。因此在第二种 DDAG 这种的组合形式相较于全部使用 eager syncrnization 和全部都使用 lazy synchronization 来说都是更好的。

基于这些定义，我们可以进一步的定义 ‘‘可融合的 DDAG 子图’’ 这么一个概念。每一个可融合的 DDAG 子图是一个 DDAG 中连续的一部分，并且可以使用一个 IterOnVec 或者一个 MxM(T) 操作实际执行。换句话说，每一个可融合的 DDAG 子图是一个连续的 DDAG 子图其包括 (1) 连续的向量操作；或者 (2) 连续、共享同

一个输入矩阵、但是不构成依赖关系的 MxV(T) 操作。拿图 5.3 中的例子来说，其中所有的向量操作可以组成一个可融合的 DDAG 子图，并且可以由一个 IterOnVec 操作完成执行。

### 5.5.4　优化算法

前文中有提到，由于 KASEN 拥有严格的编程模型，用它可以很容易地将程序的数据依赖关系变成一个表示依赖关系的数据依赖有向无环图（DDAG）。因此，一个简单的基于 KASEN 模型的优化器就可以通过枚举所有 DDAG 的等价转化形式，优化各个 DDAG 子图，计算优化后的开销这一方式求得最优的结果。不过，这样的优化方式的复杂度是随着 DDAG 中点的数量指数级增长的，因此不适合于大的程序。为了解决这一问题，我们提出了一个基于贪心的次优解求解方法。简单的说，我们的优化算法每次尝试扩展出一个尽可能大的可融合的 DDAG 子图。具体方法为尽可能的向已有的可融合的 DDAG 子图中添加尽可能多的边和点。当然前提条件是添加之后这一可融合的 DDAG 子图依然没有违反我们在前文第 5.4.3 节中定义的性质。如果用伪码的形式的话，我们的优化算法可以描述为：

**Step 1:** 初始的结果为一个空的可融合 DDAG 子图。

**Step 2:** 通过不断地添加“储备好被执行的”且“可以被融合的”点和边的方式扩展当前的可融合的 DDAG 子图。

- 一条指令如果满足下列条件之一的话它就暂时还没有准备好被执行：
  - (a) 其读取的数据还没有准备好；
  - (b) 数据准备好了但是数据的存储状态与操作的组合不满足表 5.1 中所列出的组合的情况；
  - (c) 这一语句是否被执行条件依赖于一个还未被决定的选择肢。
- 满足一下两个条件中的任意一个的两个操作是可以互相融合的：
  - (a) 它们都是只操作 Vector 类型变量的操作，且对应的数据被协同划分了；
  - (b) 或者它们都是 MxV(T) 操作，同时共用了一个输入的 Matrix 变量且互相不构成依赖关系。

**Step 3:** 插入显示的存储状态转化原语；

**Step 4:** 找出在执行完这个可融合的 DDAG 子图后依然会被使用的变量，为他们准备输出用的缓冲区；

**Step 5:** 将这一可融合的 DDAG 子图融合成一个 IterOnVec 或者 MxM(T) 操作执行。

**初始化**　为了实现上述的优化算法，我们的优化器需要首先对原程序进行一次数据流分析[64]。对于源程序中的每一条指令来说，我们的数据流分析需要输出两种不同的信息：（1）一种是 always-dependent 信息，其表示这一指令必然会依赖的指令集合；（2）另一种是 may-dependent 信息，其表示在某些执行路径下这一指令会依赖但是在其他的某些执行路径下不会依赖的指令集合。由于目前还有没有被计算出来的选择肢条件等原因这一判断还不能在现在做出。举例来说，在算法 4 中，计算出 tmp2 这一变量的 MxV 操作与在第一行的那个 Zipwith 操作的关系就是 always-denpendent。相对的，其与在第七行的 Map 操作的关系就仅仅是 may-dependent 关系，因为 not-converged 这一选择肢条件还没有被计算出来。

**扩展当前可融合的 DDAG 子图**　有了上述生成的依赖关系，一条指令是否是可以被立即执行的这一状态的判定就可以通过观察（1）它所有的 always-dependent 依赖是否要么已经被执行了，要不已经被当前扩展中的 DDAG 子图包含了；（2）没有任何的 may-dependent 依赖；和（3）相应的操作以及对应的输入数据的存储状态的组合满足表 5.1 中给出的约束条件来判断。KASEN 的优化算法需要持续不断地将当前可立即执行并且与当前 DDAG 子图中已有的操作可以互相融合的指令添加到当前可融合的 DDAG 子图中去。每一次添加了一条新的指令之后就有可能会出现一些新的可被立即执行的指令，因此上述的过程需要迭代的执行，直到没与任何的指令可以被添加。更进一步的，我们还提供一种方式使得用户可以将某一个特定的选择肢条件设定为“假定为真”或者“假定为假”。在这种情况下一些 may-dependent 可以被转化成 always-dependent 从而增加可以被融合的指令数。地关系就不再是 may-dependetn 而是 always-dependent。在这种情况下，我们就可以将图 5.3 中展示的第五到七行以及下一迭代的第三行所包含的五个操作同时融合成一个 IterOnVec 操作进行执行。

**添加网络原语**　再将当前的可融合 DDAG 子图扩展到极限之后，需要添加一部分的网络原语使得导致当前不满足约束条件的一些数据的存储转台转化到合适的形式才能够进行下一步的扩展。举例来说，在算法 4 中，变量 tmp2 和 diff 在刚刚执行完第 4 行和第 6 行的计算操作时都是存储在 pre-reduced 状态下的。如果系统使用的是 lazy synchronization 的模式的话，这两个变量的同步（即通过一次 gather 和一次 scatter 操作将他们的状态从 pre-reduced 转化成 consistent 状态）可以分别被推迟到执行第八和下一迭代的第四行的操作之前。不过，由于在 pre-reduced 和 consistent 状态的变量上执行 Map，Reduce 和 Zipwith 操作所需的计算量较之在 partitioned 状态下的变量上执行要大，在这一情况下，虽然不能够减少通讯量的大

小，但是依然需要采用复合的通讯模式，即图 5.3 (b) 中展示的方式。从表 5.1 中给出的约束条件我们可以发现，对于每一个计算的操作其输入的变量中最多只可以有一个是存储在 pre-reduced 状态下的。从这一点出发我们可以推导得到这样一个结论：最佳的通讯原语插入方法必然是只在这个可融合 DDAG 子图的最上层或者最下层插入。相对的在图中间插入一个通讯原语进行存储状态的转化并不会有任何的好处。举例来说，如果我们关心的是 $Reduce(F_1, MxV(F_2, F_1, V_C, M_P))$ 这一组合操作的最终结果，那么使用 lazy zyncronization 的方法可以帮我们节省掉对 MxV 操作产生的中间结果，一个向量的同步。我们只需要同步最终 Reduce 操作产生的那个 Shared 类型变量就可以了，这显然是比较廉价的。相反，在某些算法的运行过程中，一个 pre-reduced 状态下的 Vector 变量可能会被用于计算出多个 Vector 类型的变量。在这种情况下使用 eager synchronization 就可以减少被同步的向量的数量，从而减少通讯量。因此，我们实际的优化器会枚举所有可能的通讯模式，并用公式估测出这种情况下的开销，最终选取最小的那一个。

**准备缓存区**　在第四步中，我们需要识别出当前扩展出来的可融合 DDAG 子图在执行完毕之后有哪些产生的变量是会被后续的计算任务用到的。这些会被用到的变量需要准备相应的缓冲区用于存储它们。相对的，其余产生的变量仅仅是中间变量，通过我们的循环融合方法可以把它们优化掉。识别的方法非常简单，只需要再一次的借用我们在第一步中产生的数据流分析结果。如果一个变量会被后续的某条指令依赖（无论是两种依赖关系中的哪一种都算数）那么他就是一个输出变量。相对的没有被任何后续操作依赖的变量就是中间变量。由于需要多少的缓存区可以在执行这个可融合 DDAG 子图之前就被计算出来，所以对缓冲区的重用是非常简单的。只需要在每次执行的开始前进行适当的扩充就可以了。

**执行**　最后一步，按照定义，每一个可融合的 DDAG 子图都可以被融合成一个 IterOnVec 操作或者一个 MxM(T) 操作。所以在最后一步我们需要做的仅仅是将用户给出的函数组合成一个 X 入 Y 出的函数并用对应的内部操作进行执行就可以了。

### 5.5.5　实例研究

最后，让我们再一次的借用算法 4 中所给出的那个 PageRank 算法来展示一遍我们的优化酸的完整执行过程。在最开始的时候，由于所有的变量都处于 consistent 的初始存储状态下，位于算法 4 中第一行的 MxVT 操作可以立即执行。但是，因为没有任何一个其他的操作可以和这个 MxVT 操作融合，所以我们的优化算法产

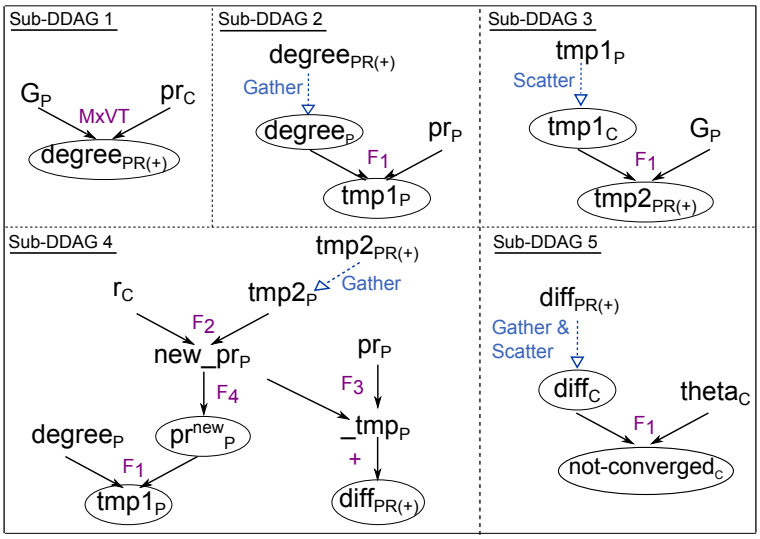



图 5.4　算法 4 的优化结果。

生的第一个可融合 DDAG 子图仅仅包含一个操作。类似的，同样由于没有可以与之融合的操作的原因，我们的优化算法产生出的第二个和第三个可融合的 DDAG 子图给也都只包含一个操作。它们和第一个可融合的 DDAG 子图的唯一区别在于它们都被自动的在 DDAG 子图的最上方插入了一个通讯原语用于保障程序执行的正确性。比如说，一个 gather 操作被自动的插入到第二个可融合 DDAG 子图的最上方。其目的为将 degree 向量由原来的 pre-reduced 状态转化到 partitioned 状态。因为用户在执行这个 ZipWith 操作时给出的 ZipWith 函数 $F_2$ 和原来的归一函数 + 不满足 pushdown law。和之前的三个可融合 DDAG 子图不同，第四个产生的可融合 DDAG 子图一次性包含了五个不同的向量操作。通过将这些操作融合成一个 IterOnVec 操作进行执行，我们的优化器极大地提升了程序的局部性，从而增加了系统的 cache 利用率。同时我们需要注意到的事，在这一可融合 DDAG 子图之中用来讲 tmp2 变量转化成 partitioned 状态的 gather 操作事实上是可以推迟的。只不过我们的优化器成功的判定出这种 lazy synchronization 的办法对程序的效果不但没有好处反而有坏处，所以才自动的采用了一种组合的方式。同时还需要注意到的是，最后的那个事实上位于算法 4 中第三行的 ZipWith 操作之所以能够添加到这个可融合 DDAG 子图中的原因在于我们将 not-converged 这个选择肢变脸设定为了“假定为真”的，因此原本的 may-dependent 关心变成了 always-dependent 关系。最后的第五个可融合 DDAG 子图也只包含了一个操作，比较简单，因此这里就不再赘述了。

## 5.6　估算公式

KASEN 模型的严格化定义不仅仅使得我们可以对用其写出的程序进行有效的分析和优化，还使得我们可以定量地估测出一段程序的执行效率。这一特性让我们可以很方便的估算出我们的优化方法到底优化了多少的内存读写、网络通讯、和内存分配。在这一节中我们将分别列出可以用于估算每一个外部和内部操作在实际执行的过程中所需要的内存读写、网络通讯、和内存分配的公式。由于这些公式的具体取值仅仅依赖于输入数据的性质和划分的方法，它们的值在执行的过程中是不会发生改变的。

具体来说，对于每一个 Vector 类型的变量，如果其在划分之后拥有 N 个 master（即原向量一共拥有 N 个元素）和 M 个 slave（即划分后为了保证 MxV（T）操作的本地执行新增了 M 个元素副本），那么（1）M 个元素会在 gather 或者 scatter 操作的执行过程中被传输；（2）如果这一变量被存储在 partitioned 状态下，N 个元素大小的内存会在执行一次本地计算操作的过程中被读取或者写入；（3）如果这一变量被存储在 consistent 或者 pre-reduced 状态下，N + M 个元素大小的内存会在执行一次本地计算操作的过程中被读取或者写入。对于 MxV（T）操作来说，每执行一次 "$ResV = MxV(F_Z, F_R, V, M)$" 操作，矩阵 M 中每一个非零元都意味着 $sizeof(M[i][j]) + sizeof(V[i]) + sizeof(ResV[j])$ 数据的内存读取和 $sizeof(ResV[j])$ bytes 数据的内存写入。

上述公式的应用范围仅为提供给用户使用的外部 API。对于两个内部操作来说，它们的内存读取和写入量可以直接通过计算输入和输出变量的个数来得到。每一次内部操作的执行都只会读取任意一个输入变量一次，并写入任意一个输出变量一次。因此，如果有一些原来存在的中间变量被通过循环融合的方式消除掉了，对于它们的读取和写入就被省略掉了。当然，这一推断成立的前提是，对于每一个下标 i，其对应的操作空间大小可以完全容纳到系统的 cache 的范围内，不会有数据在计算的过程中被逐出 cache。在一个可融合的 DDAG 子图中，每一个没有入边的节点所代表的变量就是一个输入变量。相对的，每一个有被后续操作依赖的变量就是一个输出变量。由于输出变量没有办法通过这个 DDAG 子图中本身的信息判断得出，所以我们在图示中直接将它们用圆圈标记了出来。

举一个简单的例子，图 5.4 中的第四个可融合的 DDAG 子图一共包含两个 Map 操作，两个 ZipWith 操作，和一个 Reduce 操作。根据上述的公式，如果不对他们进行循环融合，其一共需要读取 7N 元素大小的内存，写入 4N 元素大小的内存才能完成执行（N 代表这些向量的长度）。相对的，在将它们融合成一个 IterOnVec 操作进行执行之后，就只需要读取 4N 元素大小的内存，写入 2N 元素大小的内存。

这意味着大约 4N 元素大小的内存读写被我们的优化器省略掉了。

## 5.7　实现细节

在前文的叙述过程中，我们多次强调我们的 KASEN 优化算法和具体的实现是解耦合的。换句话说，我们的优化算法的效果在任意一种满足 KASEN 给出的定义的实现上产生的效果在比例上都是一样的。且这一效果可以直接通过我们在上一节中描述的数学公式计算得到，不需要进行实际的测试。不过，实现一个具体的系统并将其与现有的系统进行对比依然是有意义的。因此，我们基于 C++11 实现了一个可以执行 KASEN 程序的原型系统。其在通讯方面基于的是目前主流的一个 MPI 库，名字是 MPICH2。

我们实现的 KASEN 执行引擎是完全对称化的。这意味着在集群中没有一个计算节点充当 master 的作用，所有的节点在职责方面都是等价的。在开始执行的时候，这些计算节点会从 NFS 中各自独立的读取数据中不相交的一部分。由于这一初始的状态往往不会满足我们在前文中所描述的协同划分的需求，因此紧接着这些计算节点会进行一次全局的 shuffler 操作将数据分配到相应的节点之上。具体来说，这一 shuffler 操作分为三个步骤：（1）每一组大小相同的 Vector 和 Matrix 变量用同样的划分方法进行划分。正如我们在前文中提到的，我们会使用 hybrid-cut[11] 这个算法划分稀疏矩阵。相对的，对于稠密矩阵和向量，我们直接简单的将数据划分成连续且均等的几块[5]；（2）在划分完成之后，一次全局的通讯被用于将各个数据元素发送给它被分配到的计算节点；最后（3）我们系统中的各个计算节点会根据他们被分配到的矩阵的情况为相应的向量设立数据副本，具体方法参见前文第 5.4.1 节。

在数据初始化和载入完毕之后，计算的执行就是通过运行我们在前文第 5.5.4 节中描述的优化算法完成的。这一优化算法每一次执行的结果是产生一个可融合的 DDAG 子图，我们的系统会负责调用相应的内部操作将其代表的计算执行。KASEN 系统的用户理论上可以用包括静态函数、函数指针、functor 在内的任意一种方法定义它们的自定义函数，我们的系统会自动的将它们作用于每一个向量和矩阵的元素之上。举例来说，为了定义一个 Map 操作的函数，用户需要给出一个类型为 “*std::function<void(void *in, void *out)>*” 的函数。从中我们可以看到，输入和输出的变量都是以指针的形式传给这个函数的，它们的值分别会是一组对位的向量元素的地址指针。在得到这一系列的用户自定义函数之后，将它们融合成一个 X 入 Y 出的函数的过程非常简单：（1）对于一个可以融合成 IterOnVec 操作的可融合 DDAG 子图来说，其具体的执行过程可以用一个两层循环来实现。这一辆

从循环的外层循环负责对向量的下标进行迭代。相对的，这一两层循环的内存循环则是以拓扑顺序的方式对这个 DDAG 子图中的操作进行迭代；（2）对于一个可以融合成 MxM(T) 操作的可融合 DDAG 子图来说，其具体的执行过程非常类似一个 GEMM 操作，也可以用一个两层循环实现。同样是外层用于迭代数据，内层用于迭代用户给出的自定义函数。

至于系统的通讯部分，gather 和 scatter 操作都可以简单用一个 MPI_Alltoallv 操作实现。每一个计算节点都会在初始化阶段记录下关于“我需要给其他节点发送哪些信息，和我需要从其它节点收集哪些信息”的数据并将它们保存下来。因此每一次的 gather 和 scatter 操作都仅仅需要执行一次 MPI_Alltoallv 操作。

## 5.8　性能测试

为了测试我们的优化器的效果，我们将上述的实现和一个简单的“孤立的执行每一个计算操作，严格按照程序员给定的顺序执行，永远使用 lazy syncronization”的标准实现进行了对比。在本节中，我们首先利用前文第 5.6 节中给出的公式定量地对我们的优化算法所能达成的性能优化进行了分析。结果包括具体的我们能够节省多少内存读写、网络通讯和内存分配。随后，我们实际实现了这些算法并利用了我的系统对这些性能优化实际上能够带来多大的加速比进行了测试。最后，我们将我们的系统和已有的基于矩阵的计算框架的性能进行了比较。

### 5.8.1　定量分析

**PageRank**　正如我们在前文第 5.5.5 节中展示的那样，PageRank 算法的每一轮迭代在 KASEN 系统下实际执行的时候是被融合成了两个操作执行的。一个就是图 5.4 中可融合 DDAG 子图 2 所代表的 MxM。另外一个则是图 5.4 中可融合 DDAG 子图 4 所代表的一个 4 入 3 出的 IterOnVec 操作。因此，如果我们分别使用 N 和 S 代表一个 Vector 类型的变量在实际运行过程中所拥有的 master 和 slave 的数量。M 代表输入矩阵中非零元的数量。对于未被优化的 PageRank 算法，其每一轮迭代的执行都需要进行 $60(N+S)+24M$ byte 的内存读取和 $40(N+S)+8M$ bytes 的内存写入。其中我们假设每一个 degree 向量中的元素为 int 类型，占用 4 bytes 空间。其余所有的向量元素都是 double 类型，占用 8 bytes 空间。而对于矩阵元素，其中每一个非零元我们都是用的 COO 的格式进行存储的，需要两个 int 类型变量，共占用 8 bytes 空间。相对的，在融合优化之后，我们的程序在每一轮迭代中仅仅需要读取 $20N+24M$ bytes 的内存并写入 $16N+8M$ bytes 的内存。至于说内存的分配，由于其和内存的写入在数值上是一样的，因此通过融合我们一共节省了 $16N$ bytes 的

内存分配。

举例来说，Google programming contest 数据集[65] 中包含 $9*10^5$ 个点、$5*10^6$ 条边。将这一数据集用 hybrid-cut 划分到一个 16 节点的集群上去时对于每一个 Vector 类型的变量会需要建立 $3*10^6$ 个 slave。因此，在这个数据集上我们通过融合优化每一轮迭代都可以节省 134MB 的内存读取以及 204MB 的内存写入。这一节省分别占原来未被优化的程序的内存读取的 68% 和内存写入的 45%。

**Black-Scholes**　为了降低系统实现时的复杂度，很多的基于矩阵的计算框架都仅仅提供单入单出和双入单出这样两类的简单操作。虽然理论上借由这些简单的操作就可以组合出任意复杂的操作，但是如果没有一个优化器对它们进行融合优化的话这样组合出来的操作的性能有可能会比较低。举例来说，Black-Scholes 算法[66] 是金融计算时非常常用的一种计算股票价格的算法。它的一个标准实现需要用到 8 个 Map 操作和 6 个 ZipWith 操作。即使程序员有意的去进行人工的优化，由于操作种类的限制，也最多能够优化到一个 Map 操作和六个 ZipWith 操作的状态。不过，在我们的优化器的帮助下，这所有的操作都可以被融合成一个 2 入 2 出的 IterOnVec 操作。根据我们在前文第 5.6 节中的公式，这意味着我们优化掉了 90% 的内存读写（相较于人工优化后的版本的话相当于优化了 84% 的内存读写）。

**K-means**　我们的优化算法的另外一个优点在于它可以让程序员自由的写出比较易于理解的程序，而不用担心这样的程序的性能有问题。我们的优化器可以自动地将它优化成与复杂不易理解的程序一样的效率。拿 K-means[67] 这个非常常用的聚类算法举例的话，在 Kasen 模型下一个程序员可以非常简单的用比较直接的方式对它进行实现。比如说，程序员可以首先用一个 Map 操作和 k-1 个 ZipWith 操作为每一个数据点求到当前离它最近的中心点。然后再用 k 组的 ZipWith 和 Reduce 操作去计算更新后的 k 个中心点。我们的优化算法会自动地将这样一个程序优化成一个的 IterOnVec 操作。在这一优化的过程中，大约 $(8k-3)/(8k-1)$ 的内存读写被省略了。

**L-BFGS**　除上述简单算法之外，我们还对一个比较复杂的 L-BFGS 算法[62] 进行了定量的分析。L-BFGS 是当前最常用的一种优化算法，被广泛的应用于工业界。具体来说，为了计算一个包含 H 组数据和 W 个特征的优化问题，L-BFGS 算法仅仅需要记录 m 个长度为 W 的历史梯度向量就可以近似的还原出这一问题的 Hessian 矩阵，从而进行梯度下降的优化。因此，L-BFGS 算法比需要使用 $W^2$ 空间的 BFGS 算法需要的内存量要小得多。根据我们的计算，大约每一轮 LBFGS 算法的迭代过

程中我们的优化算法都能够节省 $2H+(10m+7)W$ 的内存读取和 $H+(4m+4)W$ 的内存写入。一篇广告预测论文中给出的一个实际的数据集[68] 作为例子的话，它具有 $10^9$ 组数据和 $10^9$ 个特征；其数据矩阵含有大约 23H 个非零元；并且 m 设定为 10。在这一情况下，我们的优化算法是计算可以为每一轮的迭代节省 872GB 的内存读取和 360GB 的内存写入。分别占原程序内存读取和写入的 48% 和 68%。更进一步的，通过组合使用 eager synchronization 和 lazy synchronization 这两种通讯模式，我们还可以节省掉一对 gather 和 scatter 操作。由于原程序一共需要 6 个 gather 和 scatter 操作，这一优化为我们省去了 33% 的通讯量。

### 5.8.2　性能测试

为了更进一步的测试我们的优化器实际能够产生的加速比，我们实际实现了上述的四个算法并将优化后和优化前程序的实际执行时间进行了对比。在测试的过程中，我们采用了一个有 16 台机器的集群进行性能评估。每台机器配置了英特尔至强四核 CPU（主频 2.5GHz），28GB 的内存空间，以及 Mellanox ConnectX3 InfiniBand（MT27500 @40 Gbps）的网卡。

**PageRank**　在评估过程中我们采用了斯坦福大学 SNAP 项目[45] 收集的若干真实世界图进行评测。下表 5.2 是我们的评估结果，其中展示的是在算法运行 30 轮迭代后得到的平均时间每一轮迭代的执行时间的加速比。通过表中的数据我们可以发现，我们的优化方式能够达到 1.22～2.20 倍的加速比。同时我们还可以从表 5.2 中的数据观察到，我们的加速比和使用的计算节点数成正相关这样一个结论。这是因为当计算节点的个数增加时，算法运行所需的元素副本数，即 S 的数量也会随之增大。同时，我们还可以发现，一般来说越稀疏的图我们的加速比越大。这两个现象都是和我们在上一节中所推导出来的公式是吻合的。

**Black-Scholes**　Black-Scholes 算法只包含两种运算，并且其执行不需要任何节点间的通信，因此我们的测试环境是在单机下进行。表 5.3 给出了我们在不同的数据规模下测试得到的加速比。从中我们可以看到，使用了我们的优化方式后能够达到 1.8 倍的加速比。即便是和人工优化后的程序相比，我们的优化器还是能够进一步的得到 1.5 倍的加速比。这一加速比相比我们在上一节中计算得到的 90% 的内存读写减少要少得多。但这主要是因为在计算这个算法的过程中需要执行一个 CDF 函数的求值计算。这使得这一程序的运行时间不完全取决于内存的载入速度。不过，虽然在运行时间上的加速比并没有想象中的那么多，但是我们的优化算法

表 5.2　PageRank 上的加速比。

| 数据集 | 计算节点的数量 | 加速比 |
|---|---|---|
| web-Google[65] (sparsity ≈ 6) | 1 | 1.22$X$ |
| | 4 | 1.46$X$ |
| | 16 | 1.70$X$ |
| web-ND[69] (sparsity ≈ 4) | 1 | 1.42$X$ |
| | 4 | 1.60$X$ |
| | 16 | 1.89$X$ |
| wiki-Talk[70] (sparsity ≈ 2) | 1 | 1.56$X$ |
| | 4 | 1.75$X$ |
| | 16 | 2.20$X$ |

在另外一个方面，即内存的使用上所能达成的效果还是非常明显的。在经过优化之后，Black-Scholes 算法的峰值内存占用被减少了大约 74%。

表 5.3　Black-Scholes 算法的测试结果。

| 股票数量 | 无优化 | 手工优化 | 完全优化 | 加速比 |
|---|---|---|---|---|
| *160M* | 50.12$s$ | 42.08$s$ | 29.04$s$ | $1.45X - 1.72X$ |
| *480M* | 160.77$s$ | 134.98$s$ | 83.24$s$ | $1.62X - 1.93X$ |

**K-Means**　类似于 Black-Scholes 算法，K-means 也是只用了 Vector 和 Shared 两种变量，因此变化使用的计算节点数对最终的加速比结果基本没有影响。在我们的测试中，数据会被均匀的分配到集群中的每台机器上。因此，我们在图 5.5 中仅仅展示了使用 1600 万个点作为输入数据，运行在 16 台机器上的情况。从图中我们可以看到，我们的优化能够达到 4.14∼5.82 倍的加速比。并且则以加速比和 k 的大小成正相关。如果我们进一步的扩大 k 的话还能够得到更高的加速比。

**L-BFGS**　此外我们还测试了对于 L-BFG S 算法的优化效果。在我们随机生成的一个包含 $1.3*10^8$ 样本和 $7*10^6$ 个特征，稠密度为 23 的数据集上，我们的优化器达到了 1.59∼3.54 倍的加速比。

### 5.8.3　与已有系统的性能对比

我们系统优化器的效果是和实际的实现解耦合的。但是，为了证明我们在上一节中测试得到的相对加速比并不是因为我们比较的原始系统特别的慢的原因，

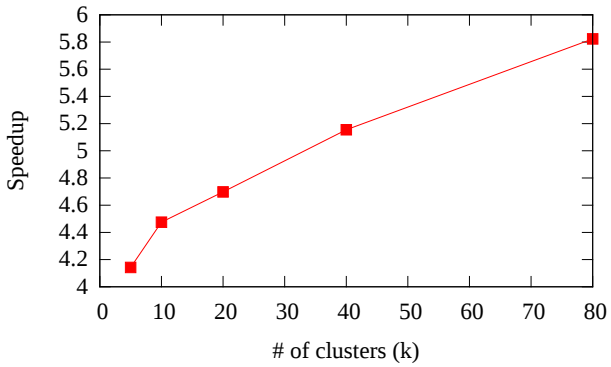



图 5.5　K-Means 算法的测试结果。

表 5.4　KASEN，Spartan[32] 和 PowerGraph[3] 地性能比较。其中所有测试均为同时使用 16 各计算节点的结果；K-Means 算法的输入点数为 160M；给想数据的单位为 ms。

| App. | PageRank | | | Black-Scholes | |
|---|---|---|---|---|---|
| | Web-Google | Web-ND | Wiki-Talk | 160M Stocks | 480M Stocks |
| KASEN | 234 | 38 | 31 | 1904 | 5201 |
| SPARTAN | 1174 | 436 | 217 | 3234 | 9023 |
| POWERGRAPH | 332 | 237 | 425 | — | — |

| App. | K-Means | |
|---|---|---|
| | 10 Clusters | 20 Clusters |
| KASEN | 115 | 190 |
| SPARTAN | 729 | 1397 |
| POWERGRAPH | 5674 | 8747 |

我们还将我们的系统性能和已有的一些系统进行了对比。具体来说，我们对比了 Spartan 和 PowerGraph 这两个系统。其中 Spartan 可以看成是一个 NumPy 的一个分布式实现，其内置了一些简单的优化方法，但是正如我们前文提到的，这些优化方法不够通用因此比我们的优化器效果要差。不过，根据 Spartan 论文中的测试结果[32]，这一系统已经比已有的一些基于矩阵的计算系统如 Presto[31] 和 SCidb[71] 等要快了。

正如我们在表 5.4 中所展示的，我们的系统较之 Spartan 系统要快得多。这一加速比的主要来源有三个方面（1）我们在数据划分采用的是更加先进的 hybrid-cut[11] 方法。相对的，Spartan 系统由于其自身的限制只能使用按快划分的划分方法。因此，我们的通讯量要远小于 Spartan 系统；（2）我们的系统中矩阵数据的非零元是按照 Hierberl Order[44] 进行存储的。因此在计算时我们的系统 cache 利用率更高。最后（3）我们的优化器效果也比 Spartan 的要更好。拿 Black-showes 算法来说的话，由于 Spartan 系统无法将 ZipWith 操作和 Map 操作融合成一个操作，其最终的

效果实际上比我们在上一节中提到的人工优化的效果还要差。至于 PowerGraph 系统，已有的很多调研[52] 都已经表明了一个基于矩阵的计算引擎是要显著的快于一个基于点的计算引擎的。我们的实验只不过是在一次的验证了这一结论。

## 5.9　本章小结

本章主要提出了一种新型的基于矩阵的编程框架 KASEN。这一框架主要解决的是原有基于矩阵的编程框架自由度太高，以至于很难进行复杂优化的问题。具体解决方法为定义严格的计算和通信模型，使得较为精确地对程序进行分析并衡量他们的表现成为可能。根据我们的评估，KASEN 的优化器可以显着地减少对内存的读/写，缓冲区的内存开销和网络流量。在我们的实验中，其最高可以将原程序加速 5.82 倍。

# 第 6 章　拓扑感知的存算融合图计算方法

## 6.1　本章概述

前文第 1.3.2 节中有提到，在提升图计算应用的速度方面一般来说有两个主要阶段的速度可以分别进行提升：（1）一个是如何尽可能快地将所需的所有数据都载入到一台机器的内存中去；（2）另一个则是如何尽可能快地将一次计算所需的数据载入到系统的 cache 中去。前者对应于我们在前文第 2.1.2 节和第 2.2 节中所提到的技术，只不过它们所需的数据分别是通过网络从其它机器中读取，以及存储在较慢的外存设备上的。而后者则是对应于我们在上一章中提到的提升程序数据局部性的方法。一般来说一个程序的数据局部性越好，它将数据载入到 cache 中的速度就越快。

不过，虽然已有的方法在上述的两个方向上都已经得到了长足的进步，但不可否认的是当前计算机系统计算的速度和系统内存的带宽存在不可忽略的差距。更糟糕的是，这一差距目前正在变的越来越大。由于摩尔定律的关系当前计算机系统的计算速度一直以指数级速度增长。相对的，内存带宽速度的提升并没有得到同样级别的提升，以至于这两者之间的差距越来越大。因此，在新型的支持存算融合的硬件[23,36] 出现之后，就有很多的研究人员寄希望于它们可以一劳永逸的解决这一问题。

事实上，我们前文中也有提到，早在十九世纪九十年代就有一些研究人员对存算融合的体系结构这一概念进行了早期的探索[33–35]。但是由于（1）给存储器件添加计算能力的工艺较为复杂难以实用；和（2）当时并没有找到特别合适的应用领域这两个方面的原因，这一早期的探索并没有取得很好的结果。但是，随着所谓大数据时代的来临，当前许多备受关注的应用类型具有内生局部性差等特点，因此急需改善存储系统的性能。同时，由于三维堆叠技术的发展，现有的比如 HMC[36] 技术可以很方便的在内存芯片上叠加计算芯片，从而达到存算融合的目的。鉴于这两个主要的问题都一定程度的得到了解决，近期有大量的研究人员重拾了对存算融合体系结构（Process-In-Memory，PIM）的兴趣。

举例来说，美国卡内基梅隆大学的研究人员设计出的 Tesseract 系统[21] 就是一个基于 PIM 技术的并行图计算加速器。它提供了一个非常类似于谷歌公司提出的 Pregel 系统[2] 的以点为中心的编程模型，因此其用户可以在 Tessercat 上很方便地进行图计算编程而不需要注意到底层 PIM 硬件的具体细节。而通过借助于 PIM 硬件的优势，他们的测试显示 Tesseract 系统较之传统的图内存图计算系统要快上

一整个数量级。不过，虽然 Tesseract 系统已经比原有的系统要快上很多了，它依然没有完全利用好 PIM 架构带来的巨大的内部聚集带宽。根据他们自己的测试，Tesseract 系统对内部聚集带宽的利用率一般来说只有不到 40% 左右，这显然是极大的一种浪费。根据我们的调研发现，造成这一浪费的主要原因在于 Tesseract 系统在执行图计算应用时产生了极大量的非本地跨 cube 通讯。而在执行跨 cube 通讯时系统没有办法利用充裕的本地带宽而是要占用相对要小得多的外部通讯带宽，因此非常容易成为系统的瓶颈。更进一步的，Tesseract 系统所采用的互联拓扑结构是一种被称之为 DragonFly 的结构。虽然这一结构较之简单的 Mesh 结构有着直径更小等优点，但它依然不是一个全对称的拓扑结构。这意味着这一结构存在瓶颈，即在某一些连接上它所需要承受的通讯流量要远超平均需要承受的流量。这一瓶颈连接的存在进一步地恶化了我们上述外部通讯所带来的问题。

为了解决这一问题，我们认为 PIM 系统的设计需要仔细的考虑并对（1）硬件的体系结构;（2）软件的运行环境；和（3）数据的划分和组织方式进行协同设计。已有的 PIM 系统也有很多注意到了数据划分的重要性，因为数据的划分决定了系统最终会产生的跨 cube 通讯的数量大小。只不过在这些系统之中，一般来说都是先决定好了系统的体系结构和编程框架之后再决定数据划分的方法的。我们之后会提到，这一决定的顺序先天的就使得很多有效的划分方法没有办法被使用。举例来说，Tesseract 系统也尝试过使用 METIS[72] 这一高级图数据划分器去划分它的图数据，寄希望于能够解决它所面临的外部通讯成为瓶颈的问题。只不过，在测试中他们发现，一共五个测试中有三个 METIS 的存在并没有产生很大的效果，另外的两个中甚至还有一个速度比原来还要慢了。考虑到 METIS 划分器本身巨大的划分开销，这样的结果显然是无法令人满意的。不过，根据我们的调研，这一结果的出现主要原因就是在于我们上面提的那个决定的顺序。许都关于图计算的研究工作都指出了系统的编程框架的选择和最终能够使用哪些图的划分算法是紧紧耦合在一起的。所以在考虑图数据的划分之前就事先将计算框架决定好的话就会造成很多有效的图数据划分方法被先天的排除在选择范围之外的情况。更进一步的，我们还发现如果将数据的划分方法和 PIM 硬件的互联方式放在一起考虑的话，我们还能够设计出有效的基于广播和数据重叠的优化方法，从而进一步的降低对外部带宽的压力。

基于以上的观察，我们设计并提出了一个新型的基于 HMC 技术的 PIM 并行图计算系统 PGIM（Process Graph In Memory）。PGIM 的硬件体系结构基本上和 Tesseract 采用的体系结构一摸一样，但是（1）和 Tesseract 提供 Pregel 系统的以点为中心编程模型点程序不同，PGIM 提供的是一个新型的名为 ‘‘两阶段点程序’’ 的

编程框架。这一框架在形式上和原来的点程序模型非常相似，但是给予了用户在选择数据划分方法上更大的自由度。在后文第 6.3.1 节中会提到，通过将两阶段点程序和一种新的划分方法 source-cut 相配合，我们的 PGIM 可以取得较之已有系统要低得多的外部通讯量。并且，source-cut 是一个非常简单的基于启发式的划分算法，所以它的划分复杂度也比 METIS 要低得多。除此之外，（2）PGIM 在通讯时利用了广播的技巧去削减前文提到的瓶颈连接的通讯量，因此其对这些特殊的连接的压力也要小得多；最后，（3）PGIM 的编程模型使得它可以非常简单的实现数据通讯和计算重叠的优化方法，从而进一步的隐藏了跨 cube 通讯的延时。

我们的这一工作较之已有的 Tesseract 系统最大的变化在于我们将 Tesseract 系统每一条被分割的边通讯一次的通讯模式变更成了副本的同步。根据我们的调研发现，虽然前一种通讯模式给予了用户更大的自由度，比如说允许同一个点在它的两条出边上写不同的值，但是这一自由度在实际的图计算应用中并没有被用到。因此，通过增强限定去掉这一自由度，我们的通讯模式可以自动地削减原来情况下会产生的冗余通讯。同样的，由于我们的副本要求同一个点的多个副本需要被同步成同一个值，因此在此基础上实现广播的优化也非常的简单。根据我们的测试结果，PGIM 对跨 cube 带宽的压力远小于已有的系统，因此具有更高的效率。

## 6.2　背景介绍

### 6.2.1　互联拓扑结构

在前文第 2.4 节中我们已经介绍了单个 HMC cube 的基本结构[36]。但是为了达成所谓“与内存容量成正比的带宽”，实际的基于 HMC 的系统都是将多个 HMC cube 互联起来构成的。简单的说，如果一个系统包含 N 个 HMC，那么它最高能够提供的内存容量就是 N*8GB。同时，它还能够提供总额高达 N*320GB/s 的内部聚集带宽。不过，虽然这样的结构可以提供这样大量的内部带宽，如何设计这些 HMC cube 之间的互联结构依然是一个非常关键的问题。

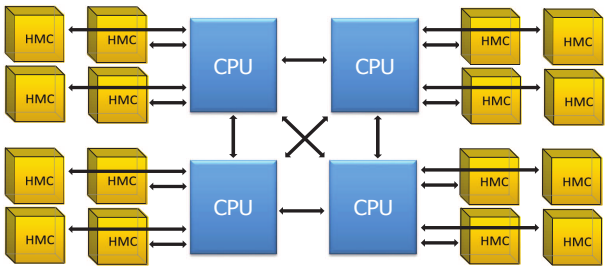
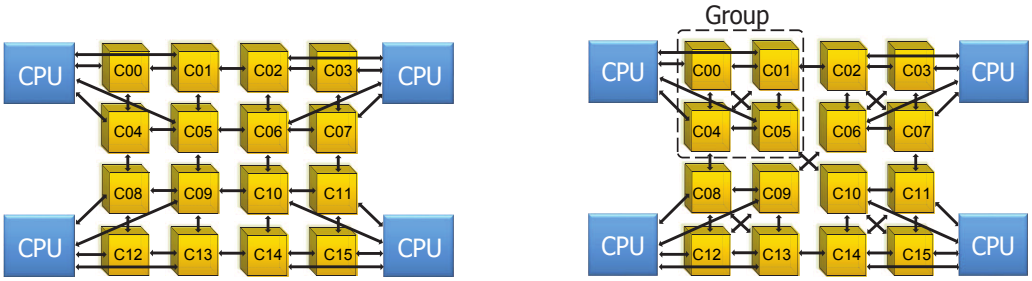



图 6.1　Processor-centric network 的互联架构[73]。

最直接的一种互联方式被称之为“processor-centric network”[73]。在这种情况下，HMC 被直接用来替代现有 NUMA 体系结构下的 DIMM 的位置。如图 6.1 所示，在这种情况下系统会使用 QPI 总线技术在所有的宿主 CPU 间达成全连接。相对的，每一个 HMC 都被指定了唯一的一个宿主 CPU 作为它的连接对象。虽然这样的体系结构非常的简单，而且基本上和现有的体系结构互相兼容，但是，根据调研发现，这样的体系结构使得系统难以利用 HMC 的高速内部聚集带宽。




(a) Memory-centric Network: Mesh　　(b) Memory-centric Network: Dragonfly

图 6.2　Memory-centric network 的互联架构[73]。

事实上，HMC 技术提供了支持基于数据包转发的路由能力，能够将一个数据包转发到指定的 HMC 或者宿主 CPU 上去。因此，基于这样的能力事实上可以设计更为复杂的 HMC 间的互联拓扑结构，从而达成更好的效率。因此 Gwangsun Kim 等研究人员就提出了一种被称之为“memory-centric network”的互联方式[73]。在这种情况下，HMC cube 之间直接通过它们的外部链接互联，并且两个宿主 CPU 之间没有任何直接联系。图 6.2 给出的就是两种具有不同的拓扑结构但是同属“memory-centric network”的体系结构。根据 Gwangsun Kim 他们的测试，这样的互联方式较之之前提到的“processor-centric network”方式能够达到最高 2.8 倍的加速比。

更进一步的，Kim 等人还测试了不同的内部互联拓扑结构。其中最常用的两种就是我们分别在图 6.2 (a) 和图 6.2 (b) 中展示的这两种被称之为 Mesh 和 DragonFly[74] 的方法。这两种方法都具有可扩展性，因此可以被运用于更大或更小的体系结构中去。根据他们的测试，他们最终推荐大家使用 DragonFly 这一互联拓扑结构。这是因为 DragonFly 较之 Mesh （1）拥有更高的内部互联程度和更小的直径；（2）可以达到和专门为 16 cube 的系统设计的名为 Falttened Butterfly[75] 的拓扑结构差不多的效率；并且（3）不像 Falttened Butterfly 拓扑那样只能用于 16 cube 的系统中，具有更好的可扩展性。在本文描述的系统中，我们也将采用图 6.2 (b) 所示的这种使用 DragonFly 拓扑进行互联的“memory-centric network”。不过事实上我们的方法理论上应该是在各种拓扑结构上都是通用的。

### 6.2.2　网络瓶颈

在单一一个 HMC 的时候，这个系统的最大外部带宽可以达到 480GB/s。相对的，内部的带宽只有 320GB/s。所以事实上在这种情况下外部带宽比内部带宽还要大。但是，由于 pin 数的限制，这一外部带宽是没有办法随着 HMC cube 数量的增加横向扩展的。因此，对于一个会同时部署多个 HMC cube 的系统来说，其内部聚合带宽一般要远高于内部带宽。就拿上图 6.2 (b) 中的 DragonFly 结构为例。由于每一个 HMC cube 最多只能向外伸出四根各自只有 120GB/s 的数据链路，所以在系统拥有 16 个 HMC cube 的情况下，是没有理论上的可能达成完全对称的全连接的。因此，实际上 DragonFly 拓扑将 16 个 HMC 的 cube 分为了四个 HMC 组，每组四个 cube。在组内，这四个 cube 是通过外部链接进行了全连接的。但是，在组外，每两组 HMC 之间仅仅只有一条外部链接进行互联。因此，跨越 HMC 组的通讯的最大带宽事实上是受到一条外部链接的带宽限制的，也就是说只有 120GB/s。相对的，由于总共有 16 个 cube，所以聚集的内部带宽高达 $16 * 320\ GB/s = 5.12\ TB/s$。这一显著的差距使得减少外部带宽的使用对整个系统的性能起到了非常关键的作用，这也间接地增强了数据划分算法的重要性。在后文的第 6.3.1 节中我们会指出，Tesseract 系统所采用的简单划分方法会产生极大的外部通讯，因此使得整个系统的带宽利用率不高。

同时，我们还需要注意到，在 DragonFly[74] 这一拓扑结构下每一条外部链接所需要承担的数据通讯量是不均等的。举例来说，如果我们假设每两个 HMC cube 之间的通讯量都是相等的话，一条链接两个不同 HMC 组的外部链接说需要承担一共 4*4=16 组的 HMC cube 间的通讯。相对的，拿链接 HMC $C_0$ 和 HMC $C_1$ 的那条外部链接来举例的话，他秩序承担 1）$C_0$ 和 $C_1$ 之间的通讯；以及 2）$C_0$ 和右上角的 HMC 组内四个 HMC cube 的通讯，即一共五组 HMC cube 间的通讯。显然，它的通讯量要比一条跨 HMC 组之间的外部链路的通讯量的 1/3 还要少。这一观察告诉我们这些跨组的外部通讯链路会比组内的外部通讯链路更早的成为系统的瓶颈。事实上，根据我们的调研，这一瓶颈的出现是不可避免的。因为其出现的根本原因在于每一个 HMC 的 cube 由于 pin 数所限只能向外提供四条外部链路。就拿图 6.2 (a) 中展示的另外一种互联拓扑 Mesh 来说的话，由于在跨越 $C_0$～$C_7$ 和 $C_8$～$C_{15}$ 的截面上只有四条外部链接，因此再怎么平均分布这四条中承受的通讯量最大的那一条也需要至少承担 8*8/4=16 组 HMC cube 之间的通讯。这一峰值的数值和 DragonFly 的情况是一样的。而且，事实上在 Mesh 拓扑下一共有八条需要承担这个级别的通讯的外部链接。相对的，DragonFly 只有 6 条。

### 6.2.3　已有的 PIM 图计算系统

正如我们在前文第 2.4 节中提到的，目前应用 PIM 技术的例子中最重要的一类就是所谓的大数据应用。之所以这一类应用非常适合于 PIM 架构的原因主要有三个：（1）基于 3D 堆叠技术的 PIM 架构可以提供非常高容量的内存。这使得大数据应用的数据有可能可以被完全地放在内存之中；（2）HMC 系统的内部聚集带宽可以随着使用的 HMC cube 的数量横向扩展，因此非常适合于大数据应用这种数据局部性不好因此需要很大的带宽的应用；（3）目前已有的大数据系统提出了非常多很好的数据编程模型，比如 MapReduce[7]，GAS[3] 等。得益于这些编程模型的良好抽象，只要我们的 PIM 系统能够提供一个类似的编程接口就可以向上层的用户屏蔽底层 PIM 硬件的细节，从而方便用户的使用。

**算法 5** 用 Tesseract 接口实现的 PageRank 算法。

```
1  ...
2  count = 0;
3  do {
4    ...
5    list_for (v: graph.vertices) {
6      value = 0.85 * v.pagerank / v.out_degree;
7      list_for (w: v.successors) {
8        arg = (w, value);
9        put(w.id, function(w, value) {
10             w.next_pagerank += value;
11           }, &arg, sizeof(arg), &w.next_pagerank);
12     }
13   }
14   barrier();
15   ...
16 } while (diff > e && ++count < max_iteration);
```

作为大数据应用中最重要的一类，目前已有很多 PIM 系统尝试去提升大规模并行图计算的效率。比如说，前文中我们数次提到的由卡内基梅隆大学的研究人员提出的 Tesseract 系统就是一个基于 PIM 的并行图计算加速器。它提供给了用户一些简单的接口，用它们可以非常方便地进行类似于 Pregel 系统的以点为中心的编程。算法 5 给出了使用 Tessercat 系统的接口实现 PageRank 算法的例子。其中的主程序就是一个两重循环（从第五行到第十三行）。在这一两重循环的外层，这一程序会遍历这一图上的所有点。而对于每一个点来说，其内存循环迭代其所有的边并对每一条边执行一个 put 函数。这一 put 函数的类型为 *put(id, void* func, void* arg, size_t arg_size, void* prefethc_addr)*，其功能就是以 arg 为参数在编号为 id 的 HMC cube 上执行一个 RPC 的函数 func。

具体来说，对于图中的每一个点，算法 5 给出的算法会首先利用第六行的程序计算其均分到每一条边上的那一份 page rank 的值。在这之后，每一条边都会对应执行一次用户自定义的 put 函数，将相应的放在这条边上的那一份 page rank 的值累计到这条边的汇点的点权上（第十行）。由于 Tesseract 系统做了特殊的优化，这一函数会以异步的形式执行，从而隐藏一部分的跨 HMC cube 访问造成的延时。最终，一个 barrier 函数被执行（第十四行），用来告知系统去等待所有的异步 RPC 执行结束。

我们从上述的例子中可以看到，虽然是由一系列的简单接口组合起来的，但 Tesseract 的编程方式基本上和 Pregel 系统所提供的点程序模型的接口是一样的。因此，Tesseract 所提供的编程模型具有很好的可编程性。不过，由于系统中实际部署了 16 个 HMC cube，因此整张数据图需要被划分分配到这 16 个 cube 上去。当一条边的两个点被划分给了两个不同的 HMC cube 的时候，这条边对应的 put 函数所执行的就是一个远程的 RPC，需要涉及外部链路的通讯。由于每一次这样的远程 RPC 通讯都以阻塞的方式去执行的话性能会非常差，所以 Tesseract 系统会自动的把这些函数转化成异步的 RPC 的方式去执行。同时 Tesseract 系统还设计了一些预取的机制去进一步的加速这一过程。不过，上述的所有优化技巧都仅仅能够被用来隐藏跨 cube 通讯的延迟，不能够降低实际的通讯量。因此如果真正的瓶颈在于带宽而不是延迟的话，上述的方法的效果就不怎么明显了。而经过我们的调研发现，由于 Tesseract 系统所采用的数据划分方法效果不佳，在实际的情况中往往就是外部链接的带宽，尤其是那些瓶颈外部链接的带宽会成为影响系统性能的真正瓶颈。因此，Tesseract 系统对整个 PIM 体系结构的内部聚合带宽的利用率一般不超过 40%，依然具有很大的优化空间。而且，根据他们自己的测试，即便是使用 METIS 对数据划分进行优化，也没有起到非常好的作用。

## 6.3　PGIM 系统

为了解决上一节中所描述的已有 PIM 系统对内部聚合带宽的利用率不高的问题，我们提出了一种新型的，专门为了并行图计算设计的 PIM 系统，名为 PGIM。在 PGIM 的设计过程中，我们很仔细地同时考虑了（1）硬件的体系结构；（2）软件的编程模型；和（3）数据的划分和组织方法这三个方面，并进行了协同设计。在本节中，我们将详细的描述我们提供的编程模型和使用的数据划分方法。事实上，从中读者可以很容易的发现，我们的编程模型实际就是为了使用我们的数据划分算法而特别设计的。同时，为了更进一步的减少 PIM 系统所需的跨 cube 的通讯量，尤其是那些瓶颈外部链接上的通讯量，我们还提出了使用广播技术的优化

技巧。最后，我们还讨论了在我们的模型下如何将通讯和计算重叠起来，从而达到隐藏跨 cube 通讯的延迟的目的。

### 6.3.1　两阶段点程序

虽然直接在 HMC 所提供的底层接口上进行编程是可能的，但是对于编写上层应用的程序员来说，让他们去直接操作这些底层接口显然是一件非常没有生产效率的事情。因此，目前的所有 PIM 系统都会提供一套较为高层的编程接口以供程序员的使用。甚至有些还会直接提供一套专门面向某一种类的应用的编程模型。这些上层的接口可以通过掩藏底层硬件细节的方式保证其可编程性，从而提升生产力。但是，我们想要强调的是，编程模型的定义事实上不仅仅影响其可编程性的好坏，还直接决定了能够选择的数据划分方法的范围。就拿 Tesseract 系统支持的类 Pregel 型的编程接口[2] 来说，它先天的就决定了用户只能使用以点为粒度的，即一维的数据划分方法。换句话说，对于一共有 16 个的 HMC cube 来说，它们中的每一个都会被分配一个互相不相交的点的集合。因此，如果一条边的源点和汇点被分配到了不同的 cube 之上，那么它们这条边上执行的那个 put 函数就会需要是一个远程的 RPC 调用。根据我们的测试结果，这一简单的数据划分方法会产生极大的通讯量，以至于成为系统性能的瓶颈。

因此，为了解决上述跨 cube 通讯量大的问题，我们在 PGIM 系统中所采用的是一种新型的编程框架，叫做“两阶段点程序”，以及一种新型的图数据划分算法，叫做 surce-cut。和以点为粒度对数据进行划分的一维划分方法不同，我们的 source-cut 是一种二维划分算法，即其对图数据的划分粒度是边。而且，和 METIS 中采用的复杂划分方法不同，我们的 source-cut 是基于简单的启发式规则的。具体来说，我们的 source-cut 仅仅是简单的将一条边（u,v）划分给节点 hash(v)%N，其中 hash 为任意的一种哈希函数，N 为系统中 HMC cube 的个数（在我们的系统中 N=16）。从上述公式中我们可以看到，虽然 source-cut 可以保证一个点的所有入边都被分配到同一个 HMC cube 之上，但是它对这个点的出边会被分配给那个节点没有作出任何保证。因此，在这种情况下没有办法使用以点为中心的传统点程序模型进行编程。为了保证程序的顺利执行，我们为这些点数据设立的相应的副本，并且会让系统自动地对这些副本进行同步。

具体来说，在 sour-cut 下，如果一条边 (u,v) 被分配给了 HMC i，那么所有和这条边共享同一个汇点，即满足形式 (*,v) 的边，也都会被分配给这个 cube。相对的，和这条边共享一个源点，即满足形式 (u,*)，的边却不一定会被分配到这个 cube 上。为此我们规定，对于每一条边 (u,v)，其所分配到的 cube 需要建立一个 u 的副

本用于后续的计算。在我们的情况下，每一个点 u 的所有入边都会被分配给编号为 hash(u)%N 的这个 cube，所以自然是这个 cube 负责对这个点进行更新。在更新完了之后这个 cube 还需要将新的权值发送给相应的副本用于数据的同步。

很显然上述的通讯模式和传统的点程序是不同的，因此我们为了适应这一新的通讯模式提出了一种新的编程模型，叫做“两阶段的点程序”。两阶段的点程序的执行过程就和它的名字暗示的一样分为两个阶段:（1）第一个阶段叫做 Generate，其负责遍历这个点的所有入边并产生一个针对这个点的权值的更新。由于在 source-cut 下一个点的所有入边一定是被分配到了同样的一个 HMC cube 之上的，所以很显然 Gnerate 阶段的执行并不会需要涉及跨 cube 的通讯。（2）在第二个阶段 Apply 的过程中。这一在前一阶段被产生的更新会被广播给该点的所有副本用于进行权值的更新。通过将原本完整的点程序有意的切分成 Generate 和 Apply 这样的两个阶段，我们的两阶段点程序模型将原来“每条边一次通讯”的通讯模式改变成了“每一个副本一次同步”。我们将在后续的论文中展示，这样的模式改变先天的消灭了一些数据传输中的冗余，因此能够达成更小的跨 cube 通讯规模。

**算法 6** 用 Tesseract 接口在两阶段点程序模型下实现的 PageRank 算法。

```
1 list_for (r: graph.all_replicas) {
2   r.pagerank = r.next_pagerank;
3   r.value = 0.85 * r.pagerank / r.out_degree;
4 }
5 list_for (v: graph.vertics) {
6   update = 0;
7   list_for (e: v.incoming_edges) {
8     update += e.source.value;
9   }
10  list_for (r: v.replicas) {
11    put(r.id, function(r, arg) {
12          r.next_pagerank = arg
13        }, &update, sizeof(update), &e.next_pagerank)
14  }
15 }
16 barrier();
```

举例而言，算法 6 给出了 PageRank 算法的另外一种实现，它使用的是（1）和 Tesseract 一样的 API；（2）效果上也和算法 5 中展示的 PageRank 实现是一样的；但（3）模仿的是我们的两阶段点程序而不是传统 Pregel 的只有一个阶段的点程序。在这个程序中，第一步与前文算法 5 类似，程序将会遍历每一个点并求出当前状态下理应分配给每一条边的 page rank 值。只不过由于我们的系统每个点都包含一系列的副本，所以在我们的程序中需要遍历所有副本而不仅仅是所有的点。接下

来的过程依然是一个两层的循环，并且这一两层循环的外层也依然是遍历图中的每一个点。只不过，在我们的编程模型下对于每一个点来说它并不是直接遍历所有的出边讲对应的 page rank 值用 put 函数发送出去，而是先遍历所有的入边求出这个点在这一轮迭代更新后的新的 page rank 值，然后再将这个值同步给它的各个副本。事实上，由于这个点的出边有很多可能没有被分配到同一个 HMC cube 上，所以找我们在数据划分模式下想要执行普通的一阶段的点程序是不可能的。相对的，由于这个点的所有入边都被分配给了同一个 HMC cube，之前迭代入边的过程是不会产生任何通讯的。唯一的通讯仅仅会在最后的副本同步上产生。

**算法 7** 用 PGIM 接口在两阶段点程序模型下实现的 PageRank 算法。

```
1 GenUpdate(incoming_edges) {
2   update = 0;
3   list_for (e: incoming_edges) {
4     update += e.source.value;
5   }
6   return update;
7 }
8
9 ApplyUpdate(v, update) {
10   v.pagerank = update;
11   v.value = 0.85 * v.pagerank / v.out_degree;
12 }
```

从上面我们可以看到，事实上我们的两阶段点程序也是可以用 Tesseract 系统提供的基础接口实现的。只不过这一实现比较繁琐，而且会需要 Tesseract 内部维护一些他目前没有维护的数据结构，如每个点的副本列表等等。因此，为了进一步方便用的使用，我们实际上提供了一组更加高层的，专门针对图计算的编程模型。在我们的编程模型下，用户仅仅需要编写两个函数，然后其他的比如副本同步等问题都会由系统自动处理。具体来说，用户需要编写的两个函数包括（1）一个 GenUpdate 函数，其输入为一个点的所有入边的集合，而输出是产生的对于这个点的更新值；以及（2）ApplyUpdate 函数，其输入为一个点和一个之前由 GenUpdate 函数产生的更新，用户可以自由地用这两个值去计算出新的点权。在每一轮迭代的过程中，GenUpdate 函数会对每一个点执行一次。相对的，ApplyUpdate 会对每一个副本执行一次。同时我们还可以发现，GenUpdate 和 ApplyUpdate 这两个函数都是可以仅仅在本地执行的，他们的运行过程中并不需要涉及跨 cube 的网络通讯。换句话说，在我们的模型下，唯一允许的通讯模式是从点到其副本的同步操作，没有任何其他种类的通讯。从后续的论述中我们可以发现，这一限制条件在没有影

响我们的编程模型的可编程性的基础上极大地增加了我们的优化余地。通过施加这样的限制，我们可以很简单的消除掉数据通讯中的冗余，从而降低通讯量。算法 7 所展示的就是在我们的 PGIM 系统中实现 PageRank 算法的代码。

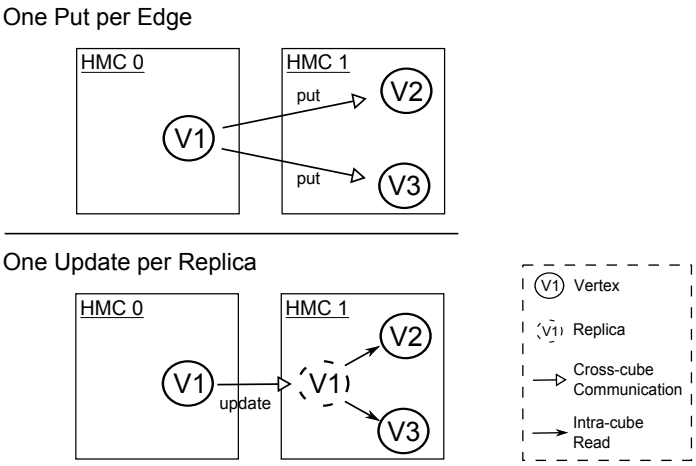



图 6.3　两阶段点程序消去冗余的原理。

为了解释为什么我们的两阶段点程序可以降低通讯量，图 6.3 给出了一个简单的例子。在图 6.3 中的这个数据图上，一共只有 V1,V2,V3 这三个点和 (V1,V2) 和 (V2, V3) 这两条边。我们假设我们选取的哈希函数 hash 的结果是 $hash(V1)\%2 = 0$，$hash(V2)\%2 = hash(V3)\%2 = 1$。在这种情况下，如果我们使用的是随机的一维划分的话，最终产生的结果就是如图 6.3 中的上半部分所展示的那样。由于一共有两条跨 cube 的边，所以需要执行两次远程的 RPC 函数调用。但事实上，这两个不同的远程 RPC 调用在很多情况下的参数都是一摸一样的，因为他们来自于同一个源点。这就显然的有数据上的冗余。不过，由于原来的编程模型允许程序员在不同的边上进行不同的操作，所以虽然事实上这一特性没有被用到，但是我们依然没有办法自动的优化掉这一冗余。相对的，在我们两阶段点程序模式下，这一冗余被自动地发现并消除。因此，我们仅仅需要一次的副本同步，比原来的通讯量要少了一半。显然，Tesseract 本身提供的 API 能够提供比我们的编程模型更好的自由度，理论上也就可以实现更多种的程序。但是，根据我们的调研，这样的自由度在图计算领域中的用处并不是特别的大。绝大多数能够用原来的编程模型表示出来的图计算应用也能够用我们的编程模型表示出来。

### 6.3.2　广播

由于每一个 HMC 地 cube 仅仅能向外提供四根外部数据链路，因此在互联的拓扑结构中不可避免地会存在某些链路承受的通讯量要比其他链路要大的情况。

因此，这些不均衡的通讯分布情况就会使得部分的链路成为整体系统的瓶颈所在。对于我们和 Tesseract 使用的 DragonFly 拓扑来说，这一瓶颈就是两个 HMC 组之间的互联链路。

根据我们在前文第 6.2.2 节中的分析，跨 HMC 组的链路所需要承受的通讯量要远高于一条普通的组内链路。虽然我们的新型划分算法可以同时降低这两者的数据量，但是无法改变这样的趋势。因此，即便是在我们的划分模式下，这些组间的链路依然会成为系统的瓶颈所在。为了降低这一问题带来的危害，我们进一步的设计了一种基于广播的优化技巧。通过广播，我们可以消除在传统点对点通信中产生的数据通讯冗余的问题。由于我们的两阶段点程序模型将通讯的模式限制为仅仅能够进行副本的同步，所以可以很明确的知道，对于一个点的不同副本来说，它们需要收到的更新是相同的。因此，对于每一个点来说，我们仅仅需要为每一个 HMC 组送去一份更新就可以了。具体来说，对于每一对 DragonFly 拓扑中的

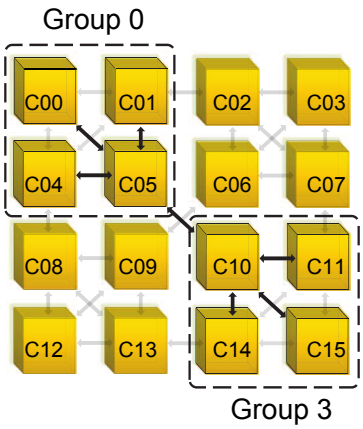



图 6.4　广播树的建立。

HMC 组，我们都会建立一个如图 6.4 所示的广播树。在图 6.4 给出的这种情况下，HMC $C_5$ 和 HMC $C_{10}$ 分别作为它们所属的 HMC 组的代理去收集和分发需要的更新。具体来说：（1）$C_5$ 会收集所有需要从左上角的 HMC 组发送给右下角的 HMC 组的更新，即满足 $\{v:(u,v)\in(C_{00}\cup C_{01}\cup C_{04}\cup C_{05})\}$ 条件的点这一轮中的更新。每一个点仅仅需要一次；（2）$C_5$ 将它收集到的更新通过链接 $C_5$ 和 $C_{10}$ 的跨 HMC 组链路传输给 $C_{10}$；最后 3）$C_{10}$ 负责将收到的更新分发给组内的 HMC cube。显然，通过这一方式，假如有一个点 v 同时在右下方的 HMC 组中的多个 cube 上具有副本，其原本会被进行多次的副本同步被归并成了一次，从而极大地减少了跨 HMC 组链路上的通讯量。

### 6.3.3　计算和通讯的重合

在具有多个 HMC cube 的体系结构中，将远程跨 cube 通讯的延迟隐藏起来是非常重要的一件事情。为了解决这一问题，现有的 PIM 系统，比如说 Tesseract，会使用异步通讯的方式将通讯和计算重叠起来。这意味着对于一个用 Tesseract 接口编写的程序来说，它们在执行完一个 put 函数之后不会等待这个函数的实际执行，而是最终等到 barrier 函数的时候一起等待。为了实现这一目的，Tesseract 系统在 HMC 的逻辑层上实现了一个缓冲机制，它会等到缓冲了一定数量远程 RPC 之后再一次性地将它们发送出去。类似的，在我们的 PGIM 系统中我们也实现了一个异步更新发送机制，不过是以一种更加简单的方式实现的。之所以比较简单，还是因为我们的编程模型限制了通讯模式到仅仅支持副本同步的地步。具体来说，在我们的系统中，每一个 HMC 的 cube 都会拥有一个本地的消息缓存区，用于接收从其它的 cube 传送过来的点的更新。在每一轮迭代的过程中，我们要求每一个 cube 依次去遍历它所被分配到的点。在每一个点对应的 GenUpdate 函数之后对应于这一点的更新就可以立即发送给对应的 cube 了。

## 6.4　测试

在本节中，我们将描述我们对 PGIM 系统的测试结果以及其和已有系统的比较。具体来说，我们的测试主要分为三个方面，一是关于测试我们的新型划分算法对通讯量的影响；二是关于我们的广播技术对瓶颈链路上通讯量的减少；最后则是整体的性能比较。在本节中，我们主要使用的数据集为表 6.1 中所示的三个。对于其中的无向图，在实际使用的过程中，我们均将其通过在两个方向上都添加一条边的方式扩展成了有向图来使用。它们均是由斯坦福大学所主导的 SNAP 项目[45] 收集并公开的。

表 6.1　PGIM 测试所用的数据集。

| 数据集 | 类型 | 点数 | 边数 |
|---|---|---|---|
| LiveJournal | 有向图 | 4,847,571 | 68,993,773 |
| Orkut | 无向图 | 3,072,441 | 117,185,083 |
| Twitter | 有向图 | 41,652,230 | 1,468,365,182 |

### 6.4.1　划分对通讯量的影响

表 6.2 给出了在这几张实际的图上进行不同种类的数据划分最终产生的跨 cube 数据通讯量。其中（1）Random 那一列的数值代表使用随机的划分方法分

配点的集合最终产生的数据通讯量；（2）METIS 那一列则表示使用 METIS[72] 这一高级划分方法进行一维划分最终产生的数据通讯量；（3）Source-cut 为 PGIM 实际产生的跨 cube 间通讯量。其中，我们假设在原本的模型下每一个边上的 put 函数说需要传递的参数大小和我们的 GenUpdate 函数产生的更新大小是相等的。对于每一种情况，我们都同时汇报了在所有的一共 16*16=256 组不同的跨 cube 通讯的平均值和最大值。

从表中的数据我们可以看到，对于随机的划分方法来说这 256 组跨 cube 通讯间的差值不大，即便是最大的那一组来说也不过是比平均值要高上 10% 左右。同时，虽然 METIS 能够极大地降低平均的跨 cube 通讯，但是其产生的结果机器不均匀。以至于在部分情况下最大的那一组跨 cube 通讯往往比随即划分的方法还要高。这一结果揭示了为什么 Tesseract 系统的性能没有办法通过使用 METIS 进行提升。因为这一最高的通讯量会成为系统的瓶颈。同时，METIS 划分的开销非常的大，表 6.2 中没有给出 Twitter 数据的结构就是因为即便是在我们的拥有 1TB 内存的大机器上也没有办法用 METIS 对这一数据集进行划分。其运行会因为内存不够而中断。

表 6.2　两个 HMC cube 间的通讯量。

| | Random | | METIS | | Source-cut | |
|---|---|---|---|---|---|---|
| | 平均值 | 最大值 | 平均值 | 最大值 | 平均值 | 最大值 |
| Orkut | 457,754 | 470,959 | 187,532 | 843,270 | 107,206 | 109,706 |
| LiveJournal | 269,506 | 289,519 | 79,224 | 352,341 | 107,789 | 115,594 |
| Twitter | 5,735,801 | 6,374,486 | Failed | Failed | 1,079,390 | 1,105,802 |

相对的，从表 6.2 中的数据我们可以看到，我们的 source-cut 方法可以极大地降低平均和最大的通讯量。和随机地一维划分相比的话，我们的划分方法在最大值上可以降低 2.5× ∼ 5.9× 的通讯量。更重要的是，我们的划分方法不会碰到 METIS 中出现的不平衡的问题，因此我们也不会出现部分外部链接先到瓶颈从而阻碍整个程序的执行的情况。

### 6.4.2　广播对瓶颈链路通讯量的影响

下表 6.3 展示了在不同数据集和不同的划分方法下的最大组内链路通讯量和最大组间链路通讯量。从中我们可以看到，一般来说组间链路上所需承受的通讯量在组内链路的三倍左右，这与我们在前文第 6.2.2 节中的分析是基本一致的。同时，我们还可以看到，虽然我们通过采用新型的数据划分算法 source-cut 同时减少了组内和组间通讯量，但是组间通讯远高于组内通讯的趋势并没有被改变。不过，

通过使用广播的技巧消去组间通讯内的冗余，实际的 PGIM 系统中组间通讯的最大值仅仅比组内通讯高上不到 46%。这显然是很大的一个提升。

表 6.3　各链路上的通讯量。

| | Random | | Source-cut | | |
|---|---|---|---|---|---|
| | 组内链路 | 组间链路 | 组内链路 | 组间链路 | 广播优化后的组间链路 |
| LiveJournal | 1,356,868 | 4,310,361 | 539,420 | 1,720,628 | 741,284 |
| Orkut | 4,610,150 | 14,684,612 | 785,578 | 2,517,454 | 734,026 |
| Twitter | 31,057,455 | 93,735,469 | 5,454,025 | 17,350,242 | 7,923,780 |

## 6.5　本章小结

本章提出了一种新型的在基于 HMC 技术搭建的 PIM 系统上进行大规模并行图计算的方法。本章描述的系统 PGIM 综合考虑了（1）硬件的体系结构；（2）软件的编程模型；和（3）数据的划分和组织方法这三个方面的因素。其提供的两阶段点程序模型可以与 source-cut 这一数据划分算法相配合，从而减少跨 cube 的通讯量。在此基础上，PGIM 还设计并实现了广播和计算通讯重叠等技巧，进一步减少了通讯的开销。

# 第 7 章　总结与展望

## 7.1　论文主要工作总结

由于图这一数据结构出色的表达能力，众多现实生活中的问题都可以用图的形式来建模与计算。而随着大数据时代的来临，被计算机系统收集到并需要加以处理的图数据的规模也正在以极快的速度不断增长。因此，提升计算系统处理大规模图数据的能力成为近期企业与学术界研究的一大重点[1–5]。

围绕这一问题，本文主要研究了如何通过提升数据载入速度的方式加快大规模图计算应用的方法。之所以选定这一主题，主要是因为图计算应用本身的性质。正如我们在前文第 1.3.1 节中所描述的，图计算应用具有高访存计算比和局部性不好等特点。因此，CPU 计算的速度基本不会成为其性能瓶颈。相反，通过提升数据加载的速度，可以很明显地加速程序的运行。

具体来说，本文分析了图计算应用在不同环境下的数据载入路径，并将其分为如图 1.1 所示的四个阶段。对这四个不同阶段的数据载入方法，本文分别作出了如下的研究：

（1）对于在分布式环境下通过网络将数据载入到本地内存的这一阶段，本文提出了一种新型的三维数据划分方法。经过我们的调研发现，已有的图计算数据划分方法都假定图计算系统所处理的数据图中每一个点和边都是不可划分的单元。因此，他们都将图计算问题的任务划分等价成了图的划分问题，从而导出了所谓的一维和二维划分方法。但是，在实际的应用场景中，有许多的数据挖掘和机器学习任务在被建模成图计算问题之后具有可以被进一步划分的点权和边权。在这种情况下，我们可以对总任务进行三维地划分，即将一张数据图中的每个点进一步的划分成子点并把同一个点划分出的不同子点分配给不同的计算节点。根据我们的实验结果，三维的划分方法可以极大地减少集群中各个计算节点之间的通讯量（在特定情况下最高可以减少超过 90% 的通讯量），从而提升系统整体的性能。

（2）对于在外存图计算环境下通过外存带宽将数据载入到内存的这一阶段，本文提出了一种分层的外存图计算数据组织格式。与分布式图计算的情况类似，这一分层的图计算数据组织格式也是利用了部分图计算应用中点权可以被进一步划分的特点，从而对它们进行了分层地组织。通过将点权分层，我们提升了一次性能够载入的点的数量，因此减少了每一个点的平均载入次数。根据我们的测试结果，通过选取一个合适的层数从而达成这一增一减之间的平衡，我们的新型数据组织格式可以极大地减少外存读写的数量。

（3）对于在将数据从内存载入到系统 cache 的阶段，本文提出了一种基于矩阵的图计算引擎的自动优化方法。这一方法在已有的基于矩阵的图计算引擎的基础上，进一步地利用循环融合等技术提升了程序的局部性。具体来说，本文提出了一种新型的编程模型 KASEN。KASEN 通过定义了一个严格的计算和通信模型，使得分析程序并衡量他们的表现成为可能。基于这样的功能，我们设计了一个 KASEN 的自动优化器，可以显着地减少内存的读/写，缓冲区的内存开销和网络流量。根据我们的测试结果，这一优化器最高可以达成 5.82 倍的加速比。

（4）最后，对于新型的存算融合体系结构，我们提出了一个考虑了其拓扑结构因素的存算融合图计算编程框架 PGIM。在设计的过程中，PGIM 综合考虑了（1）硬件的体系结构；（2）软件的运行环境；和（3）数据的划分和组织方式三方面的因素，并进行了协同地设计。因此，其最终会产生的跨 cube 通讯量，特别是在瓶颈链接上的通讯量，远低于已有的系统。

## 7.2　未来工作展望

如上所述，本文主要针对的是图数据载入的各个阶段进行的优化研究。在上层的图应用方面，为简单起见，一律假设的是输入数据为静态图数据的情况。因此，在本文的基础上，未来还可以进一步地进行下面两个方面的研究。

（1）首先是关于动态图计算的研究。在现实生活中，由计算机采集到的图数据经常是动态变化的[76,77]。比如说社交网络数据，每时每刻都会有新的边被建立和删除。虽然可以通过定期导出的方式将动态图转化成静态图，但是在数据处理的时延上这一方法有很大的劣势。在计算资源丰富的前提条件下，企业很可能希望可以尽可能快的将数据的变化反映到图计算应用的最终输出之上。因此，在静态图计算系统的基础上，还需研究如何处理边的增加和删除等情况。可以的话，希望部分图的改变不会使得原来的计算结果全部都失效，而是可以进行增量的计算。不过，由于图计算应用的多样性和复杂性，这一问题的挑战很大，可能需要针对不同种类的图计算应用分别进行适配。

（2）除动态图之外，还有一类特殊的图数据是时序图。这一类图在已有的点和边构成的拓扑关系的基础之上添加了时间戳的概念，因此更加的复杂[78,79]。同时，由于这一类图数据一般都具有极大的直径，因此现有的图计算系统在其之上的计算效率一般都不高。不过，由于这一模型可以很方便的用于建模通讯日志等自然带有时间戳的数据，其重要性不可忽视。为了提升这一类数据上的处理速度，有必要设计和实现专门的系统。

# 参考文献


[1] Low Y, Bickson D, Gonzalez J, et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow., 2012, 5:716–727.

[2] Malewicz G, Austern M H, Bik A J, et al. Pregel: A system for large-scale graph processing. 135–146.

[3] Gonzalez J E, Low Y, Gu H, et al. Powergraph: Distributed graph-parallel computation on natural graphs. 17–30.

[4] Gonzalez J E, Xin R S, Dave A, et al. Graphx: Graph processing in a distributed dataflow framework. 599–613.

[5] Buluç A, Gilbert J R. The combinatorial BLAS: design, implementation, and applications. IJHPCA, 2011, 25:496–509.

[6] Ching A, Edunov S, Kabiljo M, et al. One trillion edges: Graph processing at facebook-scale. Proc. VLDB Endow., 2015, 8(12):1804–1815.

[7] Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. Commun. ACM, 2008, 51(1):107–113.

[8] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2012. 2–2.

[9] Shaposhnik R, Martella C, Logothetis D. Practical Graph Analytics with Apache Giraph. 1st ed., Berkely, CA, USA: Apress, 2015.

[10] McCune R R, Weninger T, Madey G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv., 2015, 48(2):25:1–25:39.

[11] Chen R, Shi J, Chen Y, et al. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. 1:1–1:15.

[12] Kyrola A, Blelloch G, Guestrin C. Graphchi: Large-scale graph computation on just a pc. Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2012. 31–46.

[13] Roy A, Mihailovic I, Zwaenepoel W. X-stream: Edge-centric graph processing using streaming partitions. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2013. 472–488.

[14] Zhu X, Han W, Chen W. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2015. 375–386.

[15] Sundaram N, Satish N, Patwary M M A, et al. GraphMat: High performance graph analytics made productive. Proc. VLDB Endow., 2015, 8(11):1214–1225.

[16] Shun J, Blelloch G E. Ligra: A lightweight graph processing framework for shared memory. Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA: ACM, 2013. 135–146.

[17] Nguyen D, Lenharth A, Pingali K. A lightweight infrastructure for graph analytics. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2013. 456–471.

[18] Black B, Annavaram M, Brekelbaum N, et al. Die stacking (3d) microarchitecture. Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA: IEEE Computer Society, 2006. 469–479.

[19] Raoux S, Burr G W, Breitwisch M J, et al. Phase-change random access memory: A scalable technology. IBM J. Res. Dev., 2008, 52(4):465–479.

[20] Lee B, Zhou P, Yang J, et al. Phase-change technology and the future of main memory. IEEE Micro, 2010, 30:131–141.

[21] Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing. Proceedings of the 42Nd Annual International Symposium on Computer Architecture, New York, NY, USA: ACM, 2015. 105–117.

[22] Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing. SIGARCH Comput. Archit. News, 2015, 43(3):105–117.

[23] Gokhale M, Lloyd S, Macaraeg C. Hybrid memory cube performance characterization on data-centric workloads. Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, New York, NY, USA: ACM, 2015. 7:1–7:8.

[24] McSherry F, Isard M, Murray D G. Scalability! but at what cost? Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, Berkeley, CA, USA: USENIX Association, 2015. 14–14.

[25] The Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it /challenge9/download.shtml.

[26] Valiant L G. A bridging model for parallel computation. Commun. ACM, 1990, 33(8):103–111.

[27] Yahoo! AltaVista Web Page Hyperlink Connectivity Graph. https://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[28] Roy A, Bindschaedler L, Malicevic J, et al. Chaos: Scale-out graph processing from secondary storage. Proceedings of the 25th Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2015. 410–424.

[29] Hilbert Curves. http://datagenetics.com/blog/march22013/index.html.

[30] Abadi M, Barham P, Chen J, et al. Tensorflow: A system for large-scale machine learning. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2016. 265–283.

[31] Venkataraman S, Bodzsar E, Roy I, et al. Presto: Distributed machine learning and graph processing with sparse matrices. Proceedings of the 8th ACM European Conference on Computer Systems, New York, NY, USA: ACM, 2013. 197–210.

[32] Huang C C, Chen Q, Wang Z, et al. Spartan: A distributed array framework with smart tiling. Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2015. 1–15.

[33] Gokhale M, Holmes B, Iobst K. Processing in memory: The terasys massively parallel pim array. Computer, 1995, 28(4):23–31.

[34] Kogge P M, Bass S C, Brockman J B, et al. Pursuing a petaflop: point designs for 100 tf computers using pim technologies. Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers '96., Sixth Symposium on the, 1996. 88–97.

[35] Kogge P M. Execube-a new architecture for scaleable mpps. Proceedings of the 1994 International Conference on Parallel Processing - Volume 01, Washington, DC, USA: IEEE Computer Society, 1994. 77–84.

[36] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.1. 2015.

[37] Hybrid Memory Cube Specification 2.1. http://www.hybridmemorycube.org/specification-v2-download-form/.

[38] McSherry F. Spectral partitioning of random graphs. 529–537.

[39] Chen R, Shi J, Zang B, et al. Bipartite-oriented distributed graph partitioning for big learning.

[40] Jannach D, Zanker M, Felfernig A, et al. Recommender systems: an introduction. Cambridge University Press, 2010.

[41] Zhou Y, Wilkinson D, Schreiber R, et al. Large-scale parallel collaborative filtering for the netflix prize. 337–348.

[42] Anderson E, Bai Z, Bischof C, et al. LAPACK Users' Guide. Third ed., 1999.

[43] Zhang M, Wu Y, Chen K, et al. Measuring and optimizing distributed array programs. Proc. VLDB Endow., 2016, 9(12):912–923.

[44] Bender M A, Brodal G S, Fagerberg R, et al. Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model. 61–70.

[45] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/.

[46] Brozovsky L, Petricek V. Recommender system for online dating service.

[47] Celma O. Music Recommendation and Discovery in the Long Tail. Springer, 2010.

[48] Coates A, Huval B, Wang T, et al. Deep learning with COTS HPC systems. 1337–1345.

[49] Awara K, Jamjoom H, Kanlis P. To 4,000 compute nodes and beyond: Network-aware vertex placement in large-scale graph processing systems. 501–502.

[50] Khayyat Z, Awara K, Alonazi A, et al. Mizan: A system for dynamic load balancing in large-scale graph processing. 169–182.

[51] Hoque I, Gupta I. Lfgraph: Simple and fast distributed graph analytics. Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, New York, NY, USA: ACM, 2013. 9:1–9:17.

[52] Satish N, Sundaram N, Patwary M M A, et al. Navigating the maze of graph analytics frameworks using massive graph datasets. 979–990.

[53] Then M, Kaufmann M, Chirigati F, et al. The more the merrier: Efficient multi-source graph traversal. Proc. VLDB Endow., 2014, 8(4):449–460.

[54] Seo S, Yoon E J, Kim J, et al. HAMA: An Efficient Matrix Computation with the MapReduce Framework. 721–726.

[55] MathWorks I. Matlab and statistics toolbox release 2012. natick: The mathworks, 2012.

[56] R Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[57] Qian Z, Chen X, Kang N, et al. MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud. 197–210.

[58] Walt S v d, Colbert S C, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 2011, 13(2):22–30.

[59] Belter G, Jessup E R, Karlin I, et al. Automating the generation of composed linear algebra kernels. 59:1–59:12.

[60] Zhang Y, Yang J. Optimizing i/o for big array analytics. Proc. VLDB Endow., 2012. 764–775.

[61] Forum M P. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[62] Liu D C, Nocedal J. On the limited memory BFGS method for large scale optimization. Mathematical programming, 1989. 503–528.

[63] Lugowski A, Alber D, Buluç A, et al. A flexible open-source toolbox for scalable complex graph analysis. 930–941.

[64] Lerner S, Grove D, Chambers C. Composing Dataflow Analyses and Transformations. 270–282.

[65] Leskovec J, Lang K J, Dasgupta A, et al. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, 2009, 6:29–123.

[66] Black F, Scholes M. The pricing of options and corporate liabilities. The journal of political economy, 1973. 637–654.

[67] MacQueen J, et al. Some methods for classification and analysis of multivariate observations. Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, volume 1, 1967. 281–297.

[68] Chen W, Wang Z, Zhou J. Large-scale L-BFGS using MapReduce. 1332–1340.

[69] Albert R, Jeong H, Barabási A L. Internet: Diameter of the world-wide web. Nature, 1999, 401:130–131.

[70] Leskovec J, Huttenlocher D, Kleinberg J. Predicting positive and negative links in online social networks. 641–650.

[71] Brown P G. Overview of scidb: Large scale array storage, processing and analysis. 963–968.

[72] Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput., 1998, 20(1):359–392.

[73] Kim G, Kim J, Ahn J H, et al. Memory-centric system interconnect design with hybrid memory cubes. Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, Piscataway, NJ, USA: IEEE Press, 2013. 145–156.

[74] Kim J, Dally W J, Scott S, et al. Technology-driven, highly-scalable dragonfly topology. Proceedings of the 35th Annual International Symposium on Computer Architecture, Washington, DC, USA: IEEE Computer Society, 2008. 77–88.

[75] Kim J, Balfour J, Dally W. Flattened butterfly topology for on-chip networks. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA: IEEE Computer Society, 2007. 172–182.

[76] Han W, Miao Y, Li K, et al. Chronos: A graph engine for temporal graph analysis. Proceedings of the Ninth European Conference on Computer Systems, New York, NY, USA: ACM, 2014. 1:1–1:14.

[77] Sharma A, Jiang J, Bommannavar P, et al. Graphjet: Real-time content recommendations at twitter. Proc. VLDB Endow., 2016, 9(13):1281–1292.

[78] Wu H, Cheng J, Huang S, et al. Path problems in temporal graphs. Proc. VLDB Endow., 2014, 7(9):721–732.

[79] Zong B, Xiao X, Li Z, et al. Behavior query discovery in system-generated temporal graphs. Proc. VLDB Endow., 2015, 9(4):240–251.