学校代号 10532 学 号 S12102049
分 类 号 TP391 密 级 普通

湖南大學
HUNAN UNIVERSITY

# 硕士学位论文

# 基于单机多核系统的图处理研究

学位申请人姓名 周东伟
培 养 单 位 信息科学与工程学院
导师姓名及职称 陈浩 教授
学 科 专 业 计算机科学与技术
研 究 方 向 多核虚拟化
论文提交日期 年 月 日

学校代号: 10532
学　　号: S12102049
密　　级: 普通

湖南大学硕士学位论文

# 基于单机多核系统的图处理研究

学位申请人姓名: 周东伟
导师姓名及职称: 陈浩 教授
培　养　单　位: 信息科学与工程学院
专　业　名　称: 计算机科学与技术
论文提交日期: 年　月　日
论文答辩日期: 年　月　日
答辩委员会主席:

# Graph Processing Research on Single Multi-core Systems

by

ZHOU DongWei

B.E. (Hunan University) 2015

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of engineering

in

Computer Science and Technology

in the

Graduate school

of

Hunan University

Supervisor

Professor Chen Hao

April, 2014

# 湖 南 大 学

## 学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名:　　签字日期:　　年　　月　　日

## 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

1、保密□，在____年解密后适用于本授权书

2、不保密。

(请在以上相应方框内打"√")

作者签名:　　签字日期:　　年　　月　　日

导师签名:　　签字日期:　　年　　月　　日

# 摘　要

随着各种类型的社交网络的兴起，基于图结构数据的企业级应用正变得日益广泛与重要。而如何高效便捷的分析、调试和处理这些与日俱增的大规模图数据成为当前高性能计算领域的研究人员所面临的最迫切的问题之一。目前，已经存在一些分布式的解决方案，但是基于分布式的图处理系统依然存在着许多悬而未决的难题。从经济角度来看，在分布式系统上进行数据分析、调试和处理需要额外的资源消耗，例如计算资源以及维持计算的能源消耗。与此同时，从用户角度分析，分布式系统对于相关的开发人员也提出了较高的要求，例如，开发经验，从而增加应用成本。而就分布式系统本身而言，不同计算节点之间的消息延迟与负载均衡则很容易成为系统的性能瓶颈。于是，有相关学者和研究人员提出基于单机共享内存的图处理系统，实验证明，这些单机上图处理系统经过严谨而合理的设计不仅具有高效便捷的优势，还能大大的降低开发成本。但是，随着网络的发展，数据量的增大，单机系统在扩展性上很难满足这样的挑战。目前应用于企业级的图处理系统大部分仍然基于BSP模型的分布式系统，而现有单机系统往往从系统IO优化处着手，两者之间很难结合在一起。

本文从兼容性、容错性与便捷高效的角度考虑，提出了基于并行BSP模型的单机图处理系统GPSA。基于BSP模型的图处理过程主要有计算和消息分发两个步骤。在传统的BSP模型中，由于图数据的局部性问题，单个顶点上的计算和通信两个步骤需要顺序执行。在本文，GPSA利用Actor编程模型取代线程来改善系统的并发性和计算吞吐量；其次，结合Actor与BSP模型将图计算中顶点的计算和消息通信过程解耦，降低两个相邻的超级步之间的依赖关系，使计算和通信两个步骤以流水线的方式并发执行。另外，GPSA利用内存映射的方式来提高数据的读取和更新能力。实验证明，GPSA能够显著提升单机系统上图处理的性能。

**关键词：图处理；Actor模型；BSP模型**

# Abstract

Graph-based applications become more and more common due to the rising of all kinds of online social networks and other problems encountered in enterprise development environment. Due to the increasing need to process the fast growing graph-structured data (e.g., social networks and web graphs), analysing, debugging and developing an agile graph processing system becomes one of the most urgent problems facing systems researchers. Though some distributed approaches have been proposed, however, these approaches still have many unresolved problems. In the perspective of economy, analysing, debugging and processing large-scale graph needs extra resources, such as computing resources and to maintain calculation of energy consumption. Furthermore, it requires the developer to be skilled such as rich experiences of distributed developing, which increases the cost. In the perspective of the distributed approaches, the load balancing and the communication latency among different computing nodes are possibly to be the bottleneck of the system. Therefore, some researchers proposed approaches on just one PC. Experiments show that the single machine approaches with a rigorous and reasonable design could not only gain the reasonable performance but also can greatly reduce development costs. However, with the developing of the Internet, the amount of big data is becoming larger and larger, it is hardly to satisfy the challenge with the single machine approach. Nowadays the enterprise graph processing system is based on the BSP (Bulk Synchronous Parallel) model while the single machine approach mainly focus on IO performance, which makes the two hard to work together.

Motivated by this, in this paper, we introduce GPSA, a single-machine graph processing system based on a parallel BSP computation model by considering the compatibility, convenience and fault-tolerant. The traditional BSP based system has two main procedures: the computing and the message dispatching. Because of the locality of the graph processing, the two procedure sin the traditional BSP model are executed sequentially. In this paper, GPSA takes advantage of actors to improve the concurrent degree on a single machine with limited resource. GPSA improves the BSP computation model to fit actor programming model by decoupling the message dispatching procedure from computing procedure. Furthermore, we exploit memory mapping to improve the IO performance to avoid frequent data loading or unloading operation. We show, through experiments and theoretical analysis, processing large-scale graph on a single machine with GPSA performs well.

**Key Words: Graph processing; Actor model; BSP model**

# 目　录

# 插图索引

# 附表索引

# 第1章 引言

## 1.1 选题背景及意义

随着博客、社交网络、以及云计算、物联网等技术的兴起，互联网上的数据正以前所未有的速度在不断的增长和累积，大数据[1–6]成为当今科技的主题之一。这些大数据分散在整个网络中，数据量极其巨大，并且基于这些大数据的各种各样的新型应用也不断出现发展。基于大数据的应用也日渐凸显出巨大的经济和科研价值，大数据技术满足了信息时代社会大众对高效、准确获取信息的强烈愿望，所以社会各界都对大数据的投来关注的目光。然而，在社会大众感受到来自大数据技术的魅力的时候，进行相关大数据处理和服务的提供者则不得不面对海量数据所带来的各种压力和挑战。基于云计算平台的技术[7–9]成为人们处理大数据的选择之一。在云计算平台中也已经开展一些相关研究，并取得一些成果，例如基于Hadoop的数据处理[10]和基于日志分析的搜索引擎用户行为分析[9]等。

在大数据中存在着大量的以图为结构进行存储的数据。由于现实世界中的许多应用场景都可以利用图来表示，所以这类结构数据具有非常巨大的经济和科研价值。同时，将这些数据以图结构进行存储则有利于分析、处理并发掘处蕴含在其中对经济、科技、教育等等领域非常宝贵的信息。不仅如此，图结构的数据在其他领域也有着非常广泛而深刻的应用，在生命科学、安全、金融服务等很多领域也存在类似的数据集，例如，交通，报纸文献，用户行为分析[9]疾病爆发路径以及科学研究发表文章中的引用关系等。另外，还有许多其他图计算问题也有着重大的实际价值，如最小切割，连通分支等问题。这些大规模图数动辄就是数亿顶点，数十亿条边，数据量之大更是前所未有。相比于无结构的数据而言，图结构之间的数据的联系更加密切，使得大规模的图对象更加难以处理[11]。而如何高效的处理大规模的图对象，更加快速准确的发掘其价值，成为研究人员和研究组织争相分析和研究的热门对象。

目前，在大规模图对象处理领域虽然已经有人提出一些解决方案，例如基于云计算[12]、Map-Reduce[13,14]的方案，但是基于这些已有框架的大规模图计算在应对需求灵活多变的情况下，已经逐渐呈现出力不从心之势。于是，相关研究人员在总结参考现有框架的基础上，根据图数据的特性提出了一些基于分布式计算的图计算框架，例如，Pregel[15]，GPS[16]等，这些图计算框架很快成为处理大规模图对象的主力。然而，基于分布式的图计算框架依然存在诸多弊端和问题，例如，分布式系统中各个计算节点之间的负载均衡问题，数据交换和通信时延问题，这些问题逐渐成为分布式图处理系统的瓶颈。考虑到分布式系统固有的瓶颈

问题和在分布式系统上进行大规模图对象处理的复杂性，有相关研究人员指出在单机系统中构建经济、高效的图处理系统将会改进当今大规模图处理领域的现状。

## 1.2 相关研究现状

目前，针对大规模图处理系统的研究国内外虽然取得了一些成果，但是并没有统一的标准。本文根据系统的使用场景和规模，将这些大规模图处理系统进行分类如下：

- 分布式图计算系统
- 单机图计算系统

本节分别针对这两类的国内外研究现状进行介绍和阐述。

### 1.2.1 分布式系统

通常已有的分布式计算框架并不是完全适合图计算。Map-Reduce[13,14]是由谷歌提出的一种基于Key-Value键值对的编程模型。用户需要自定义map()和reduce()两个函数。map()函数用来处理Key-Value键值对并且负责生成一系列中间键值对，reduce()函数用来对具有相同Key的中间值进行归约。如图1.1所示，统计一个庞大的文档库中每个单词的出现次数，首先自定义map()函数读取文档库生成一系列的“单词-次数”键值对，然后使用reduce()函数对中间键值对进行规约处理，接着统计出文档库中每个单词的出现次数。

```
map(String key,String value):
    for each word w in value:
      EmitIntermediate(w,"1");
reduce(String key,Iterator values):
    int result = 0;
    for each v in values:
      result += parseInt(v);
    Emit(AsString(result));
```

图 1.1 MapReduce的单词词频统计

在分布式系统中，Map-Reduce模型可以很方便的实现大数据的并行计算。运行时系统处理输入输出、调度以及故障处理等。虽然Map-Reduce常常被用来解决大型图的问题,但是通常对图算法来说都不是最优的解决方案，也不是最合适的方案。虽然Map-Reduce对数据处理的基本模式有聚合以及类似SQL语句的查询方式等，但是这些扩展方式通常对大型图计算这种消息传递模型来说并不理想。。例如基于Map-Reduce的GBASE[17,18]、PEGASUS[19]，以及一些基于Map-Reduce的云计算平台的方案[7]和基于Hadoop的海量数据处理[10]。Hadoop是一个分布式的系

统基础架构，是Java平台上的Map-Reduce实现。朱珠[10]等人通过对现有的分布式计算和存储等关键技术进行分析，结合Hadoop的集群技术，提出一种基于Hadoop的海量数据处理模型。该模型通过需要进行关联匹配来减少网络之间的消息通信量，并且充分利用并行计算的优势，但是针对大规模的结构化的图数据而言这样的解决方案并不适合。

为解决上述问题，Google的工程师们，以BSP[20]模型为基础，提出一种全新的以顶点为中心的图计算模型Pregel[15]，真正开创了大规模图处理系统的新篇章。BSP模型作为计算机语言和体系结构之间的桥梁，又称作桥模型。在BSP模型中，计算由一系列用全局同步分开的周期为L的计算组成，这些计算称为超级步。在一个超级步中，各处理器均执行局部操作，并且可以通过选路器接收和发送消息。然后作一全局检查，以确定该超级步内的所有处理器是否已经完成操作。若是，则进行到下一个超级步，否则下一个L周期被分配给未曾完成的超级步。BSP模型强调了计算任务和通信任务的分开。此外，BSP放弃了程序局部性原理，简化程序的设计和实现。

Pregel[15]是第一个以顶点为中心的分布式图计算框架。由于Map-Reduce并不适用于这种需要大量消息传递的场景，所以需要发展新的计算模型去完成这样的任务。Pregel的计算通过一系列以顶点为核心的超级步的迭代完成，在每一次迭代中，每个图中的顶点会接收来自上一次迭代的信息，并发送信息给其他顶点，同时可能修改其自身状态以及以它为顶点的出边的状态，甚至改变整个图的拓扑结构，该迭代过程持续到整个图中所有顶点完成收敛，即所有顶点由激活状态转换为不激活状态。在第0个超级步，所有顶点都处于激活状态，所有的激活顶点都会参与到所对应的超级步中计算。顶点通过将其自身的状态设置为停止表示他已经处于不激活状态，这就表示该顶点在该超级步中没有进一步的计算需要执行。如果顶点接收到消息，那么将会被唤醒并进入激活状态，如图1.2所示。

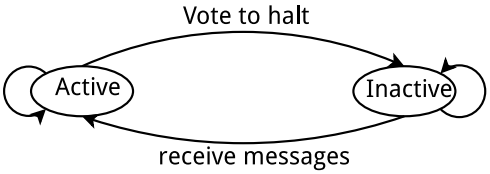



图 1.2 顶点状态机

虽然图算法也可以被写成是一系列的链式MapReduce调用，但是需要将整个图的状态从一个阶段传输到另一个阶段，这样就需要很多的通信和随之而来的序列化和反序列化的开销。采用Pregel可以将顶点和边保存在执行的那台计算机上，而仅仅利用网络传输信息，通过引入BSP中超级步的概念来避免这样的情况。Pregel提供了一套完整的API接口，并具有较好的扩展性和容错机制。

Pregel是严格的BSP模型，采用“计算-通信-同步”的模式完成大规模的图计

算。在Pregel中图处理框架中依然存在一些缺陷，如pregel以顶点为中心的，但是在很多应用中是以若干组合的顶点为中心，在这样的场景下Pregel就不适合。此外，Pregel不支持动态的负载均衡，无法在运行时进行任务的重新分配，减少通信的目的。于是，有很多学者提出了一些类Pregel的系统，并且针对上述的问题做出优化。

GraphLab[21]是为高效的机器学习算法提出的一种基于共享内存的分布式并行处理框架。GraphLab将数据抽象成图结构，将算法的执行过程抽象成Gather、Apply和Scatter三个步骤，其并行的核心思想是对顶点的切分和两端分区方案。与一般的BSP模型不同的是GraphLab总是在最近获得的数据上进行计算来保证计算过程的持续性。GraphLab的缺陷在于容错性，负载均衡等方面。Trinity[22]也是一个基于内存的分布式系统，它主要关注如何在图计算过程中优化内存使用和降低通信代价，为此Trinity需要昂贵的高带宽设备。

Mizan[23]是一个类Pregel系统，在BSP编程模型的基础上对大规模的图对象进行处理。Mizan提出了细粒度的负载均衡。首先读取数据，将数据分区并分配给不同worker。然后整个系统执行一系列的superstep，每一个superstep都被一个全局的同步栅栏分开。在每一个超级步中，每一个顶点对来自上一个超级步中的消息进行处理，并将消息发送给邻接的顶点，以供下一个超级步计算使用。与Pregel不同的地方在于，Mizan通过在worker之间移动选择的顶点来保持整个图的负载平衡。Mizan着重于负载的运行时监控和全局的分布式顶点移动管理。在运行时，Mizan针对worker发送给其他worker的消息数量、接收的消息数量以及当前超级步的每个顶点处理消息所消耗的响应时间作为主要的监控指标。如果消息数量过多或者当前计算节点在当前超级步处理消息所消耗的响应时间过长则判定为该worker负载过重，需要进行重新分配。然后，在该worker内选择不平衡的顶点，同时将所有worker根据监控指标统计的数据进行排序。第一个和最后一个，第二个和倒数第二个，以此类推，两两编为一组，并将较为繁忙的worker上选择的顶点移动到与之同组的较为空闲的worker上。但是，由于Mizan的该特点也势必导致在图计算的过程中顶点在不同的计算节点上的频繁移动，给计算效率造成一定的影响。

GPS[16]是一个具有可扩展性、容错、丰富API接口的大规模图处理系统。它不仅提供了更加丰富的API,并且支持领域特定语言Green-Marl[24,25]，同时可以在计算期间动态的重新为图分区。GPS通过合理的动态重新分区算法，极大的减少了各个计算节点之间的消息通信量。在Pregel中只有以单一节点为中心的算法才可以通过Pregel的API实现，GPS则改进这一缺陷，支持多顶点组合的算法。为减少各个计算节点之间的消息通信量，GPS提出一种LALP的分区方式，该分区方式通过将具有较高入度和出度的顶点分配到不同的节点上实现。可惜的是，GPS的

扩展性是以庞大的设备数量，高昂的经济费用为代价的，在单个计算节点上存在资源浪费和利用率不高的问题，如在内存不够时，不能充分利用磁盘等。此外，GPS不能很好地处理强连通分量的相关问题。PowerGraph[26]是一个用于处理特殊幂律分布的自然图的分布式框架。PowerGraph引入一种以顶点为依据的分区方式，支持边的重复分布式存储，并且提供类似于GraphLab的gather、apply、scatter三种操作。但是，PowerGraph如何处理动态图和非幂律分布的自然图尚未可知。

除此之外，还存在一些为特别应用而定制的一些分布式框架。Kineograph[27]也是一个类Pregel系统，该系统主要用于处理不断动态变化的图对象。它可以抓取输入中数据之间的关系，在图数据结构中表示出来，同时快速创建数据快照，并且为了保证图动态变化后图数据的一致性，将图处理的过程和图更新的过程区分开来。Little Engine[28]系统根据one-hop重复性对图进行重新分区，在整个图结构上实现负载均衡，减少网络开销。但是，该系统和Kineograph类似都是用来处理特殊问题而定制的系统，其中Kineograph主要用于从一系列流式输入的数据快照中进行快速数据挖掘的目的，Little Engine用于处理在线社交网络的扩展性问题。因此，从应用层面来看，Kineograph和Little Engine存在一定的局限性。

综上所述，分布式的图处理系统依然存在着许多悬而未决的难题。对于采用了BSP模型的分布式系统而言，整个图的处理过程是由一系列迭代的超级步组成，相邻的超级步之间需要额外的同步等待时间，而不同计算节点之间的负载均衡问题则成为分布式系统的主要性能瓶颈之一。另外，跨不同计算节点的边会导致顶点通信的延迟问题。从经济角度来看，在分布式系统上进行数据分析、调试和处理需要额外的资源消耗，例如计算资源以及维持计算的能源消耗。与此同时，从用户角度分析，分布式系统对开发人员也提出了较高的专业要求，增加应用的开发成本。

### 1.2.2 单机系统

虽然分布式系统极大的提高了图计算的效率，妥善的处理了一批亟待解决的问题。分布式框架的最大的缺陷在于分布式资源是必不可少的，同时这对开发者而言，无论是费用还是技术门槛都要求过高，并且用分布式框架开发，不方便问题定位和调试。因此，有学者就提出一些能运行于单机系统上的能够处理大规模图对象的系统。

Grace[29]是基于内存的图感知事务型图计算系统，被设计应用于要求低延迟的图计算。另外，计算机中核的数量与内存的不断增加，单个计算节点的计算能力从理论上可以替代同等配置的分布式系统，即将以往运行于分布式系统上的应

用在单个计算节点上完成。在内存中，程序的访问呈现出局部性的特点，所以Grace将整个图分成较小的子图存储在内存中，将这些子图分配给不同的核并行计算。Grace为用户提供了一套完整的查询和更新图的API，查询操作可以应用于特定的顶点或者分区，更新操作主要指添加或者删除顶点和边。但是，Grace的重点在于从图感知和事务两个方面出发，虽然考虑现代计算机多核的特点，Grace基于内存的多核并发的方式容易引发同步和死锁等问题。

Ligra[30]是一个轻量级的基于共享内存的图处理框架。该框架适用于单机多核的并行计算的图处理，使得图的基于遍历的算法简单易写。实现Ligra的图算法很灵活：针对图中边的处理，另外一种则是针对顶点。虽然Ligra充分利用了多核和并行计算的优势，在遍历算法中表现卓越。但是该框架的共享内存的特性限制其在动态图计算中的发挥，并且不适合异步计算。

Pearce[31]等学者为图遍历设计了一种异步式系统。该方案将表示图结构的稀疏矩阵经过压缩存储在磁盘上，顶点的状态信息存储在内存中，计算通过并发容器调度。但是，该系统与Kineograph和Little Engine一样都是为了特殊的应用而设计，存在一定的局限性，无法应对灵活多变的应用需求。

与传统以顶点为中心的处理方式不同，X-Stream[32]是以边为中心的共享内存的图处理框架。X-Stream[32]使用基于边的scatter-gather编程模型，而在顶点中保存处理的状态。在Scatter阶段，所有边发送边上的更新值，在gather阶段则设置边上的更新值。

GraphChi[33]是一个基于磁盘的高效图处理系统。由于个人计算机通常没有足够的内存装载整个图，所以把图存储在硬盘上，与内存相比，硬盘的数据读写速度较慢，会拖慢整个计算过程。因此，GraphChi设计了一种更快速的，减少随机读写硬盘的平行滑动窗口(PSW)。PSW使用较少次数的非顺序磁盘读写快速地从硬盘中处理边和顶点，并且支持异步计算模型。PSW处理图的过程分为三个步骤：

- 从磁盘中装载图。
- 更新顶点和边。
- 将更新的值写入磁盘。

Graphchi有着较好的扩展性和图处理性能，以较低的代价解决复杂的问题，是该领域一个成功的榜样。但是，Graphchi依然存在一些性能问题。首先Graphchi并发度有限，其次它将图处理过程分为计算过程和IO过程，造成计算的不连续。为此有相关学者提出TurboGraph，它充分利用多核的高并发的特性，使计算过程和IO磁盘访问重叠，提高计算效率，减少计算时间。

TurboGraph[34]提出一种pin-and-slide模型。该模型实现一种多向量相乘的列视图算法，并且设计两种不同类型的线程，执行线程和回调线程，使得在IO操作方

面可以通过流水线的overlapping提升图的处理效率。在操作系统中，线程虽然极大的提高了程序的并发性，但是线程的频繁的上下文切换依然是一种较为浪费操作系统资源的操作，从多核的角度考虑，TurboGraph仍然没有充分挖掘多核的优势。

## 1.3 研究内容

大规模图处理无论是在分布式环境还是在单机环境，都存在各种各样的问题。分布式图处理系统存在着负载均衡，消息通信时延等瓶颈问题，而单机的图处理系统则伴随着IO开销过大，CPU计算资源未能充分利用等问题。

本文从兼容性、容错性与便捷性的角度考虑，通过对国内外研究现状进行分析、比较与总结，针对大规模图数据处理课题提出如下几点主要研究内容：

- 研究分析现有分布式图处理系统中遗留的诸如负载均衡、通信延迟等问题。
- 研究单机多核系统中使用角色并发模型替代传统线程并发模型以提高并发度，增加系统的处理数据的吞吐量的可能性。
- 分析传统BSP模型在计算处理过程中存在的强耦合处理流程的弊端，并在角色并发的模型基础上对传统BSP并发模型进行改进。
- 分析在新模型下图处理过程中的数据行为和相关的IO操作。
- 实现一个具有可扩展性、兼容性、便捷高效，同时能够充分发挥单机多核计算能力的图处理框架。

## 1.4 论文组织结构

第一章介绍了本文的选题背景和科学意义，进而介绍了目前大规模图处理领域已经取得的研究成果并指出了它们的不足之处。

第二章首先介绍图的概念和术语及其相关的常见应用算法，然后介绍针对大规模图对象的计算模型，并发模型以及IO技术。

第三章详细介绍了GPSA系统的整体设计与实现，首先从改进传统BSP模型入手，继而分析在新模型下数据的访问行为和工作方式，最后详细介绍GPSA主要功能模块的具体实现。

第四章对本文提出的GPSA系统从可行性、效率、稳定性和便捷性四个方面进行分析和举例说明。

第五章是实验分析。首先详细介绍了实验的软硬件平台，然后介绍实验测试所使用的数据集，接下来介绍实验对比的两个单机系统上图处理系统，并用三个应在不同数据集上的实际运行效率和多核利用率进行对比、分析和验证。

最后，对全文所做的工作进行了总结，并指出了下一步的研究方向。

# 第2章　图计算相关概念与技术

## 2.1　图的相关概念

图是一种复杂的抽象数据结构，能够表示不同真实世界中不同组件之间的关系[35]。从数据结构的角度出发，图是由有穷非空的顶点集合和表示顶点之间关系的边的集合组成。假如我们使用$G$表示一个图，$V$是图中顶点的集合，$E$表示是图中边的集合，那么一个图就可以表示为：$G = (V, E)$。在图中，若顶点$V_i$到$V_j$的边没有方向，则这条边是无向边，无向边可以使用无序二元组$(V_i, V_j)$表示。如果图中任意两个顶点之间的边都是无向边，则该图是无向图，而一个图的任意两个顶点之间都有一条边，则称该图为无向完全图；若顶点$V_i$到$V_j$的边有方向，则称这条边为有向边，有向边可以使用有序二元组$(V_i, V_j)$表示，其中$V_i$表示起始顶点，$V_j$称为目的顶点。如果图中任意两个顶点之间的边都是有向边，则称该图为有向图，而有向图中任意两个顶点间都有方向相反的两条有向边，则该有向图为有向完全图。含有n个顶点的无向完全图有$n(n-1)/2$条边，含有n个顶点的有向完全图有$n(n-1)$条边。

在无向图中，一个顶点拥有的边的数量就是该顶点的度。在有向图中，顶点的出度就是以该顶点为起始顶点的边的数量，顶点的入度就是以该顶点为目的顶点的边的数目，该顶点的度为它自身入度和出度之和。在图的边上对其进行标示数字来表示该边的相关的数据信息，则该信息称为边的权重。边上带有权重的图就称为网。

在图中，若存在一系列首尾相接的边组成的顶点序列$(V_i,V_m,V_n,.....,V_j)$使得从顶点$V_i$到达顶点$V_j$，则称该序列为顶点$V_i$到顶点$V_j$的路径。其中，路径上边的数目就是该路径的长度，若是带权图，则路径的长度是该路径上边的权重之和。若一条路径的起始顶点和终止顶点相同，则称该路径为回路。若路径中只有起始顶点和终止定点是同一定点的路径称为简单回路。否则，没有相同顶点的路径称为简单路径。

如果图$G^0$的顶点和边属于图$G$则称图$G^0$是$G$的子图。在无向图中，如果存在一条从顶点$V_i$到顶点$V_j$的路径，则称顶点$V_i$到顶点$V_j$是连通的。若图中的任意两个顶点都是连通的则称该图为连通图。

## 2.2　常见图算法

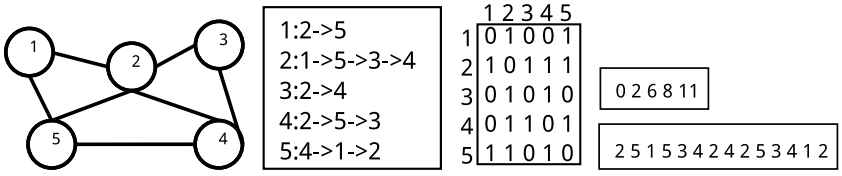
图论作为数学领域中的一个重要分支已经有数百年的历史。人们发现图的许

多重要特性，发明了许多重要的算法，其中许多困难问题的研究仍然十分活跃。在解决现实问题时，其他数据结构，例如线性表、树等，都具有明显的条件限制，而图结构中任意两个数据元素间均可相关联。在结构和语义方面，图较之线性表和树更为复杂，但是也更加具有表现能力。因此，现实世界中的很多数据往往都是以图的结构进行保存。

### 2.2.1 图的存储

在计算机中，要表示一个图$G = (V, E)$，有两种标准的方法：邻接矩阵和邻接表。这两种方法既可以表示有向图，也可以用来表示无向图。邻接表表示由一个顶点$V$和以该顶点为起始顶点的边的集合组成。对于图$G$中每个顶点$V_i$，把所有邻接于顶点$V_i$的顶点$V_j$保持为一个单链表，则该单链表称为顶点$V_i$的邻接表，如图2.1 (b)所示。邻接矩阵表示的原理就是利用两个数组，其中一个数组用来保存顶点，另外一个数组用来表示边,矩阵中为1的位置表示存在一条边，否则为0，如图2.1 (c)所示。

另外，图还可以用CSR(Compressed Sparse Row)格式进行保存，CSR格式将顶点$id$和边分别用两个数组进行保存，其中边数组保存边的目的顶点$id$并按照出发顶点$id$排序就，而在顶点数组中顶点的$id$就是数组的索引，每个顶点保存第一条边在在边数组中的索引，如图2.1 (d)所示。




(a) 5个顶点的无向图　(b) 邻接表表示　(c) 邻接矩阵表示　(d) CSR表示

图 2.1 图的表示方法

### 2.2.2 遍历图

图的遍历是指从图中给定的某一源点出发，对图中的顶点访问且仅且访问一次的操作。图的遍历算法有两种：广度优先和深度优先。

广度优先搜索是最简单的图算法之一，也是很多其他图算法的重要基础。对于给定的图$G = (V, E)$和一个确定的源顶点$s$,广度搜索算法的主要流程如下：

- 首先访问顶点$s$，并将顶点$s$标记为已访问。
- 依次搜索$s_0$所有邻接顶点$s_1,s_2,...,s_m$，并将其标记为已访问。
- 再按照$s_1,s_2,...,s_m$的次序依次访问它们的未被访问过的邻接顶点,并将其标记为已访问。
- 以此类推，直到图中所有和源顶点$s_0$有路径相同的顶点都被访问过为止，

如图2.2所示。

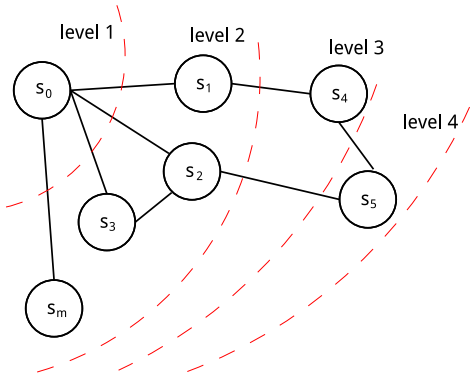



图 2.2 广度优先遍历

对于给定的图$G=(V,E)$和一个确定的源顶点$s$，深度优先遍历的基本流程如下：

- 首先访问顶点$s$，并将顶点$s$标记为已访问。
- 依次搜索$s_0$的所有邻接顶点$s_i$，若$s_i$没有被访问过，则把顶点$s_i$作为新的源点，继续深度优先遍历，直到图中所有和顶点$s_0$有路径的顶点都被访问一次为止。
- 如果图中存在其他未被访问的顶点，则另选一个作为源点，重复上述步骤，直到所有顶点都被访问过为止，如图2.3所示。

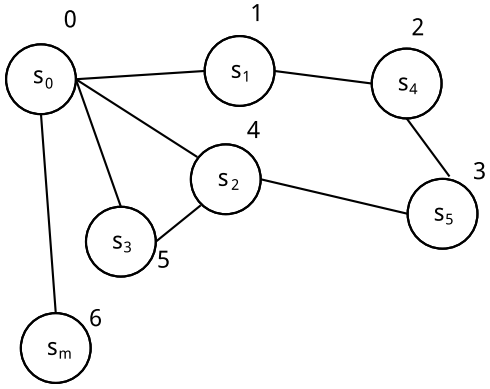



图 2.3 深度优先遍历

### 2.2.3 单源最短路径

单源最短路径是图论中寻找从开始顶点到某一终止顶点的最短路径的算法。对于已知图$G=(V,E)$，寻找从给定的源点$s$到顶点$S_i$属于$V$的最短路径。最常见的算法就是Dijkstra算法，它的主要思想是：如果路径$P$是从源点$s_0$到顶点$s_j$的最短路径，其中顶点$s_i$是路径中的一个点，则从顶点$s_0$沿着路径$P$到顶点$s_i$的路径就是从顶点$s_0$到顶点$s_i$的最短路径。假设源点为$s_0$，集合$U=s_0$，distance[i]记录从顶点$s_0$到顶点$s_i$的最短路径，path[i]记录从顶点$s_0$到顶点$s_i$的路径上的前一个顶点，则Dijkstra算法的主要流程如下：

- 从顶点集合$V-U$中选择使distance[i]最小的顶点$s_i$,并将顶点$s_i$加入集合$U$。
- 更新与顶点$s_i$直接相邻的distance值。
- 重复以上步骤，直到顶点集合$U$与图的顶点集合$V$相同，则计算完成。

### 2.2.4 最小生成树

最小生成树是指用少的边连接图中所有的顶点。最常见的最小生成树算法是Prime算法和kruskal算法。Prime算法的主要思想是从某一顶点开始，选择与它相邻的具有最小权重的边，并将该顶点加入到生成的树的顶点集合中，然后每一步都从一个顶点在生成树的顶点集中而另一个顶点不在该集合的各条边中选择权重最小的边，并将顶点加入生成树的顶点集合中。如此往复，直到图中的所有顶点都加入到生成树中为止，如图2.4所示。假设无向带权连通图$G=(V,E)$，$T=(U,ET)$是最小生成树，其中$U$是最小生成树的顶点的集合，$ET$是最小生成树的边的集合，则Prime的算法流程：

- 初始化，$ET$为空，$U=s_0$。
- 对于所有边$(s_i,s_j)$，其中顶点$s_i$属于最小生成树的顶点集合$U$，顶点$s_j$不属于最小生成树的顶点集合$U$，选择权重最小的一条边加入最小生成树。
- 重复步骤2，直到所有顶点都加入最小生成树中。

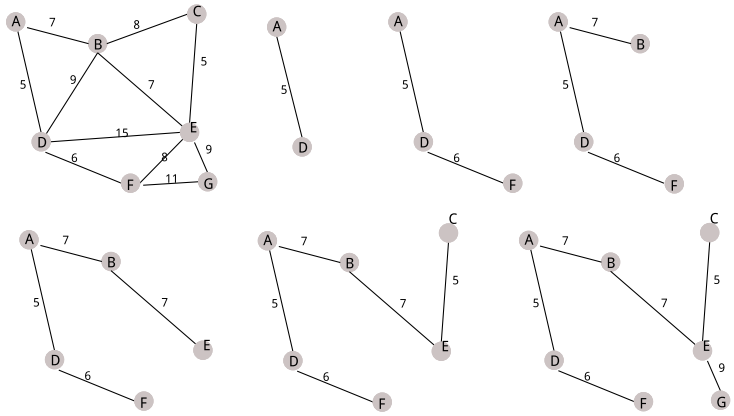



图 2.4 Prime算法示例

kruskal算法的主要思想是将图的n个顶点看做是一个含有n棵树的森林，然后从图中选择一条权值最小的边，如果该边的两个顶点属于两个不同的树，则将两个树合成一棵树，否则如果该边的两个顶点已经属于同一棵树，则不可取，一次类推，直至森林中只有一棵树为止，如图2.5所示。假设无向带权连通图$G=(V,E)$，$T=(U,ET)$是最小生成树，其中$U$是最小生成树的顶点的集合，$ET$是最小生成树的边的集合，则kruskal的算法流程：

- 初始化，n棵树的森林
- 对于所有边$(s_i,s_j)$，选择权值最小的边，如果顶点$s_i$和$s_j$属于两棵不同的树则加入，否则不可取。

- 重复步骤2，直到森林中只剩下一棵树。

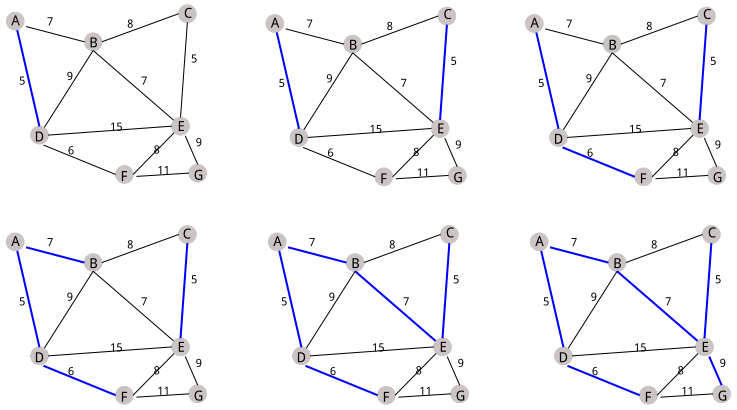



图 2.5 kruskal算法示例

### 2.2.5 PageRank

PageRank是著名的网页排名算法。PageRank是一种根据网页之间的相互连接的投票关系对网页在网络中所占的权重进行计算的一种算法。在网络世界中，网页之间通过超链接互相联系，如果把网页看做顶点，网页之间的超链接关系看做是边，整个网络都可以看成一个图。通过对各个网页在网络中的权重进行计算，有利于提高搜索结果的准确性和相关性。在PageRank中，如果网页$Page_i$存在一条指向网页$Page_j$的一个超链接，那么就说网页$Page_i$给网页$Page_j$投票。一般来说，重要的网页所指向的网页也往往是比较重要的网页，同样的高权重的网页指向一个低权重的网页，也可以使低权重的网页的权重提升。其中每个网页的初始权重为1/n，那么该网页每有一个连接指向其他网页，那么该网页对其他网页提供的投票数为1，若有n个链接指向其他网页，那么该网页对其他网页每个提供1/n的投票。反之，某个网页收到来自其他网页的投票数之和即为该网页自身的权重。如上所示，进行多次迭代，直到所有网页的权重都趋于收敛则计算结束。

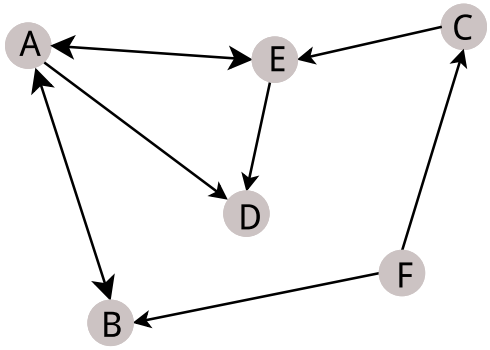



图 2.6 kruskal算法示例

如图2.6所示，一个由5个网页组成的网图关系，A分别给网页B、网页D、网页E提供权重为1/3的投票，B给网页A提供权重为1的投票，C给网页E提供权重为1的投票，D由于没有玩连接所以D默认给每个网页提供权重为1/6的投票，E分

别给网页提供群众为1/2的投票，F分别给网页提供权重为1/2的投票，构造的投票的矩阵$h$如下所示：

$$h=\begin{bmatrix} & A & B & C & D & E & F \\ A & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ B & \frac{1}{3} & 0 & 0 & 0 & 0 & \frac{1}{2} \\ C & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ D & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ E & \frac{1}{3} & 0 & 1 & 0 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} & A & B & C & D & E & F \\ A & 0 & \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \\ B & 0 & \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \\ C & 0 & \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \\ D & 0 & \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \\ E & 0 & \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \\ F & 0 & \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \end{bmatrix}$$

为了能够确保PageRank能够快速收敛结束运算，在实际进行计算时，往往需要让投票矩阵乘以一个阻尼系数，该阻尼系数一般为0.85，同时由于没有外链接的网页传递出去的权重会是0，所以需要给每个页面一个最小值，如下$G$所示：

$$G=0.85*h+\begin{bmatrix} & A & B & C & D & E & F \\ A & 1 & 1 & 1 & 1 & 1 & 1 \\ B & 1 & 1 & 1 & 1 & 1 & 1 \\ C & 1 & 1 & 1 & 1 & 1 & 1 \\ D & 1 & 1 & 1 & 1 & 1 & 1 \\ E & 1 & 1 & 1 & 1 & 1 & 1 \\ F & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} * \frac{0.15}{6} = \begin{bmatrix} 0.03 & 0.17 & 0.03 & 0.17 & 0.45 & 0.03 \\ 0.31 & 0.17 & 0.03 & 0.17 & 0.03 & 0.45 \\ 0.03 & 0.17 & 0.03 & 0.17 & 0.03 & 0.45 \\ 0.31 & 0.17 & 0.03 & 0.17 & 0.45 & 0.03 \\ 0.31 & 0.17 & 0.88 & 0.17 & 0.03 & 0.03 \\ 0.03 & 0.17 & 0.03 & 0.17 & 0.03 & 0.03 \end{bmatrix}$$

在该图中共有6个顶点，则每个顶点的权重初始化为1/6，则所有顶点的初始权重构成一个向量$x$:

$$x=\begin{bmatrix} 0.17 & 0.17 & 0.17 & 0.17 & 0.17 & 0.17 \end{bmatrix}$$

按照如下公式开始迭代，直至$x$稳定收敛：

$$x=Gx$$

最后经过计算，$x$收敛值如下所示：

$$x = \begin{bmatrix} A & 0.18 \\ B & 0.17 \\ C & 0.12 \\ D & 0.23 \\ E & 0.23 \\ F & 0.08 \end{bmatrix}$$

PageRank值决定了网页在搜索引擎中的排名，通过该算法，用户可以在搜索时获得更加准确和权威的信息，具有非常重要的实用价值。

## 2.3 大规模图计算技术

大规模图对象，顶点与边的数目都非常巨大，难以全部载入内存进行处理。由于图的算法往往呈现出一定的随机性，而这种情况在大规模图中表现的更为明显，给传统的需要一次性将图载入内存的应用带来巨大的挑战。例如，顶点$V_i$存在某一条$(V_i,V_j)$，由于该图比较大，顶点$V_i$以其相关的边已经载入内存中，而顶点$V_j$却还存在于磁盘上，此时对该图进行遍历，需要访问顶点$V_j$，则需要访问磁盘读入顶点$V_j$的相关信息，在这样的情况下传统图处理算法就会因为不得不频繁的访问磁盘而造成处理效率极度低下。另外，图的有些算法是递归实现的，对于大规模的图，这种实现很容易出现无限递归调用的情况。所以，对于大规模图的处理无论是可行性还是计算效率，传统的算法实现都不适用，所以需要研究能够用来处理大规模图对象的框架和算法。

目前，针对大规模图处理已经存在一些分布式和单机上的解决方案，例如，Pregel、GPS、Mizan、GraphChi以及X-Stream等。虽然这些解决系统各自的实现方式细节不同，但是总的来说，对于大规模的图处理技术而言，目前主要分为两种：以顶点为中心的计算模型和以边为中心的计算模型。

### 2.3.1 Vertex-Centric模型

顶点为中心的大规模图处理模型由谷歌在Pregel系统中提出。在以顶点为中心的计算模型中，顶点是整个计算的核心，每个顶点保存着图处理中的主要信息，顶点之间互相发送各自的更新消息，并且顶点可以根据接受到来自其他顶点的消息对其进行计算并更新自己的消息。同时顶点有两个状态：激活和未激活。

当顶点接收到消息的时候，顶点就处于激活状态，否则处于未激活状态。当所有顶点都处于未激活状态的时候，计算完成。以顶点为中心的计算模型非常适合大规模的图处理，这主要是因为虽然图处理的整体计算很多，但是具体分到每个顶点的计算量非常少，同时无需关心顶点是否存在于内存中或者是否存在于本地或者存在于其他计算节点。在实现以顶点为中心的计算模型中，用户只需要实现每个顶点如何处理消息的方法即可，极大地简化了基于大规模图的应用开发。例如，如果要实现一个基于以顶点为中心的最短路径算法，首先指定一个源顶点$V_0$，顶点的值表示从源顶点到某一个顶点的距离，那么源顶点的值初始化为0，其他初始化为无穷大；然后从源顶点开始将自身的值封装为消息，并发送给它的邻接顶点，其他顶点接收到消息后，将消息与自身的顶点值进行比较，若消息值较小，则将消息值加1后作为自己新的顶点值，然后将该更新状态发送给其他顶点。否则，不做任何处理。当图中所有顶点都不再更新状态，即所有顶点都处于未激活状态，那么计算结束。

### 2.3.2 BSP计算模型

Vertex-Centric模型从理论上极大地简化大规模图处理过程。但是这种没有控制的随意消息发送方式只能处理简单的图应用算法，由于需要发送和产生大量的消息，对于没有载入内存的顶点而言，或者消息的接受顶点存在于远程计算节点上，就需要对大量的消息进行缓存，这样由于需要IO操作或者通信延迟而造成的状态消息更新速度不匹配，很可能造成某些顶点的重复计算。例如，对于一个最短路径算法而言，某个顶点因为存在于内存中，它处理的消息更新迭代可能远远高于保存在磁盘上或者远程计算节点上的顶点。而对于一个基于以顶点为中心的PageRank算法，一次迭代是需要严格的消息更新状态的，在没有同步的控制下，难以保证正确性，同时容易造成消息风暴。所以为避免这样的问题，将以顶点为中心的大规模图处理过程，分为一系列连续的超级步，每个超级步结束时同步，并完成对消息的发送和缓存等操作，该模型就是BSP计算模型。

BSP是Bulk Synchronous Parallel的缩写，是一个通用的并行计算模型，最初作为一个并行计算领域中软件和硬件之间的“过渡模型”而提出的。BSP，“大块”同步模型，其概念由哈佛大学的Valiant和牛津大学的Bill McColl提出，支持消息传递，块内异步并行，块间显式同步。BSP一般由三个部分组成：处理的任务单元、任务单元之间的消息通信和对全局任务单元进行全局同步的机制。

BSP模型非常适合分布式的图计算环境。首先，BSP模型具有全局的数据通信网络。BSP中的各个处理单元之间的通信或内存的存取和大多数基于分布式内存或消息传递的并行模型是类似的。不同的是BSP的通信是一个全局的概念，不是点对点的通信，能够很好的解决图计算过程中数据的随机访问问题。其次，

BSP模型具有全局的路障同步机制。路障同步的引入则能保证图处理过程的完整性，提高整个图处理的鲁棒性和可靠性。因此，很多基于分布式内存的大规模图处理统都构建在BSP模型的基础上，例如Pregel、GPS等。

在BSP模型中，任务单元负责对分配给它的大规模图的顶点进行循环遍历，并调用消息处理函数，然后将顶点的更新消息发送给其他顶点。从顶点的角度而言，图的处理过程由两个过程组成：顶点调用消息处理函数计算消息和发送自身的更新状态。如此迭代，直到所有任务单元完成对顶点的遍历以及函数调用和消息发送过程，在全局路障处同步，当前超级步完成，所有任务单元进入下一个超级步。

### 2.3.3 Edge-Centric模型

以边为中心的计算模型[32]与以顶点为中心的计算模型较为类似，但是殊途同归并且本文不以该模型为重心，所以不做过多赘述仅对该模型进行必要的介绍。如图2.7所示，首先，该模型对边进行遍历，并且通过边将消息发送出去；然后，对边上的目的顶点进行更新。与以顶点为中心不同，以边为中心的计算模型将计算根据边分为Scatter和Gather两个阶段，在Scatter阶段对边进行读取，然后将边的起始顶点的信息通过边发送给目的顶点；在Gather阶段，统一将Scatter阶段的消息进行处理更新。

```
edge_send(edge e)
    send update over e
edge_update(update u)
    apply update u to u.destination

while not done
    for all edges e
        edge _send(e)
    for all updates u
        update_gather(u)
```

Edge-centric

```
vertex_send(vertex v)
   send updates over outgoing edges of v
vertex_update(vertex v)
   apply updates from inbound edges of v

while not done
    for all vertices v that need to send updates
        vertex_send(v)
    for all vertices v that have updates
        vertex_update(v)
```

vertex-centric

图 2.7 以边为中心和以顶点为中心的算法流程

## 2.4 并发技术

随着计算机多核处理器的出现和内存不断增大，单台计算机的计算性能实际上已经有着很大的提升。在过去40多年时间里，计算机性能一直遵循着摩尔定律，集成电路上可容纳的晶体管数目，约每隔18个月便会增加一倍，而集成电路的性能（计算能力）也将提升一倍。近年来，集成电路的集成程度已经非常高，芯片上元件的几何尺寸不可能无限制的缩小下去，摩尔定律面临挑战，遭遇瓶颈。另外，仅仅提高单核芯片的速度会产生过多的热量并且无法带来相应的性能改善。然而，人们对于电脑的要求不断提高，迫使处理器向高性能的方向发展。如果多一颗同一性能的处理器，理论上处理能力是原来的两倍。于是，为了进一

步提高性能，就需要更多的处理器，将多个处理器置入单一芯片中，构成多核心处理器。于是，有相关学者和研究人员提出基于单机共享内存的图处理系统，例如GraphChi、X-Stream等。实验证明，这些单机上图处理系统经过严谨而合理的设计不仅具有高效便捷的优势，还能大大的降低开发成本。但是，随着网络的发展，数据量的海量递增趋势已经越来越明显，现有的一些单机系统在扩展性上很难满足这样的挑战。目前应用于企业级的图处理系统大部分仍然基于BSP模型的分布式系统，而现有单机系统往往从系统IO优化处着手，两者之间很难互相兼容。另外，局限于目前多核的并发编程理论并不成熟，造成单机环境下多核计算机的计算能力并没有被充分利用。传统的单机多核并行软件开发往往是基于共享内存的，共享内存就很容易引起竞争条件，为保证程序的准确性和一致性，就必须对可能引起竞争的变量或者语句进行加锁保证互斥或者进行同步。此外，从硬件层面考虑，多核虽然提高了计算性能，但是随着核数的增多，核与核之间也会出现消息通信，内存数据一致性等问题.

常见的图算法有遍历、最短路径、最大连通分量等。对于规模较小可以完全载入内存的图，单线程的处理方式需要顺序依次对顶点进行遍历访问即可。但是，当图的规模巨大时，这种单线程的执行方式就难以满足需求。大规模图的处理技术同时具有IO密集型和计算密集型两种特点。IO密集型是因为大规模图对象数据量庞大，无法一次性全部载入内存进行计算，同时还会涉及大量的中间结果的读写。而计算密集型则是因为基于大规模图对象的应用需要对整个图进行分析计算，涉及大量的迭代和计算。因此，传统的单线程的开发方式不仅能造成IO阻塞降低图处理的效率，还无法充分发挥多核计算机的优势，造成计算资源的浪费。因此，对于大规模图对象的处理一般采用并发的方式来提高图处理的效率。

为了能够更加有效的利用硬件所提供的性能，传统的应用开发方式现在已经不适用了。以往的应用开发方式在大部分情况下所面对的都是只有一个单独的处理器，也就是意味着顺序执行的单线程应用。如今，多核计算机逐渐成为主流配置，并且价格也在不断地降低。而要充分发挥多核计算机的计算能力，最简单的办法就是利用并发，编写能够在多个核心上运行的任务，并且任务之间可以通过共享数据的方式进行通信协同工作，也就是并发程序。

### 2.4.1 大规模图处理中的并发技术

大规模图对象因其数据量之庞大，传统的顺序的完全载入内存的方式不适合处理大规模图对象。如果需要将所有图都载入内存当中进行计算，就需要使用分布式的计算资源，将图分为若干部分，均衡地分配给每一个计算节点，同时不同的计算节点之间需要互相通信以交换数据。

以顶点为中心的计算模型从概念上将顶点看做一个单独的执行单元。由于计

算机的基本调度单位是线程，即使是把图均衡的分配在不同的计算节点上，也无法满足每个顶点同时并行执行，所以需要多线程并发技术来进一步提高图处理的效率。采用多线程的并发技术，可以将图按照一定的规则分配给不同的线程，每个线程负责一部分顶点。多线程下以顶点为中心的处理流程如下：

- 线程$T_0$读取顶点$s$。
- 从磁盘读取顶点$s$的所有消息。
- 调用用户自定义的消息处理函数，并更新该顶点的状态信息。
- 线程$T_0$负责将消息放入自己的消息发送缓冲区，如果缓冲区已满，则将消息发送给顶点的邻接顶点。
- 如果顶点$s$的邻接顶点位于同一个线程上，那么该线程就需要将消息进行缓存；如果位于不同的线程$T_i$上，那么就需要进行线程之间的消息通讯，将消息发送给该线程。线程$T_i$将线程保存在自己的阻塞消息队列中，等消息队列满的时候，然后根据消息的接受者的$id$，将所有消息进行缓存，并清空消息队列。

由于消息的接受者可能位于不同的线程，所以不同的线程之间需要大量的消息通讯，为了提高效率每个线程都有自己的消息发送缓存和消息接受缓存。此外，这些消息在下一个超级步来临之前并不会被处理，所以某个线程接收到消息之后，就需要将这些消息写入到磁盘上，所以一般而言每个线程同时会设置一个读写的缓冲区。多线程的并发技术提高了图计算过程中的数据吞吐量和处理效率。但是由于目前编写正确的并发程序依然非常困难，这是因为多线程并发往往是基于共享内存的，容易出现同步、阻塞和死锁[36]。

### 2.4.2 Actor并发模型

虽然多线程的并发技术提高了图处理的数据吞吐量和处理效率，但是当线程的缓冲区被写满的时候需要引入大量而频繁的IO操作，所以会当前线程会挂起。多线程频繁的上下文切换依然是一个非常重量级的操作。另外，图计算不仅仅属于计算密集型，还属于IO密集型。尽管以顶点为中心的计算模型简化了大规模图处理的流程，但是多线程的并发技术还是引入了一定的复杂性。

Actor角色模型是一种不同于内存共享的并发技术。角色模型是用“角色”的概念，每个Actor角色都通过邮箱来异步地传递消息。角色模型中的语义概念非常贴近人们的实际生活中的概念，例如，邮箱类似于生活中的邮箱，消息发送到邮箱，人们可以从邮箱中获取邮件，类似的角色可以通过邮箱接受消息。在Actor角色模型中，主要的对象有三个：角色、邮箱和消息。角色是Actor并发模型中的执行运算的主体；邮箱则是角色接受消息的缓冲区；消息则是角色执行运算的驱动。角色的消息发送过程是异步，发往不同角色的消息不共享。消息一旦生成，

一般不能被非接收Actor修改。角色之间可以并发执行，不必顺序执行。同时由于邮箱将不同角色在内存中的地址空间分开，不存在内存共享和同步的问题，所以在角色模型中没有锁和同步块的概念，也不会出现因为同步或者死锁而引发的问题。基于以上特点，角色模型能够支持更高并发量，角色之间的切换更加轻量级，也更加安全。

在JVM平台上，常见的Actor角色并发模型实现有Scala Actor、Jetlang、Akka和Kilim等。有学者[37]对这些常见的Actor库，对各种库分别从状态封装性、安全的消息传递、调度等方面进行介绍和说明，如2.1所示。最后通过Threadring测试包对各个库进行并发度和消息分发速度等方面进行性能测试，结果表明Kilim拥有最好的效果。从以顶点为中心的计算模型的角度考虑，Actor并发模型比线程并发更加适合处理大规模图对象。Actor角色模型中，角色能够接受消息，执行计算，发送消息，从语义上非常接近顶点本身。可是，目前关于Actor并发模型在图计算中的应用并没有相关的工作，缺乏相关的可参考对比的标准。本文从探索性、简化问题的角度考虑，选择kilim作为本文的Actor并发模型。

表 2.1 JVM平台的Actor库

| 类库 | 封装性 | 公平调度 | 位置透明 | 移动性 |
|---|---|---|---|---|
| Scala Actor | 否 | 是 | 否 | 否 |
| Kilim | 否 | 否 | 否 | 否 |
| Actor Foundry | 是 | 是 | 是 | 是 |
| Jetlang | 是 | 否 | 是 | 否 |
| Akka | 是 | 是 | 是 | 是 |

### 2.4.3 Kilim简介

Kilim[36,38,39]是一个使用原生JAVA编写的角色模型的并发库。在Kilim中，主要的角色模型用*Task*类表示，邮箱使用*Mailbox*类表示。Kilim的角色并发过程主要由一个称为Weaver的后期进程来实现，该进程需要直接修改类的字节码，从而使包含有*Pausable*签名的方法在运行时能够支持调度程序的快速调度。Kilim不支持公平的调度，它的调度程序总是调度最需要对消息进行处理的角色运行。调度程序启动有限数量的内核线程。在有限的内核线程的基础上，启动更多的轻量级线程或者角色，即*Task*，每个*Task*的状态信息由*Fiber*类对其进行管理，不需要陷入内核线程，从而实现快速的上下文的切换和调度。

通过继承*Task*类可以实现一个角色，Kilim的*Mailbox*类实现消息的发送和接收消息。一般而言，一个角色需要一个邮箱，但是Kilim中角色和邮箱的实现是分离的，也就意味着，可以根据具体应用的不同而灵活的组合角色和邮箱，基

于Kilim编写代码也很简单，只需要继承*Task*类，覆盖*execute*方法，并将相应的执行逻辑放在该方法内即可。然后通过如下三个步骤完成：

- 编译: javac -d ./classes 类名.java
- weave: java kilim.tools.Weave -d ./classes 类名全称
- 运行: java -cp ./classes:./classes:$CLASSPATH 类名

## 2.5 磁盘IO

在大规模图处理中，除了需要将大量的数据到写入到磁盘中，同时图数据的访问又极易出现随机无序访问。在单机的图处理系统中，IO问题成为关注的重点，如何避免图数据的随机访问成为单机图处理系统提升效率的主要问题。目前，单机系统中针对IO的优化操作主要有两种：顺序访问和异步IO。

### 2.5.1 顺序访问

顺序访问就是按照数据存储的先后次序依次进行数据读写操作。采用顺序访问的好处在于不必把全部数据都载入内存中，只需要读取部分数据，处理部分数据即可。而在大规模图中，图的顶点之间的联系会非常的多，例如，某一条边的起始顶点存在于内存，而它的目的顶点依旧保存在磁盘上，对顺序访问带来巨大的挑战。为达到顺序访问的目的，可以对边按照起始顶点进行排序，同时把边按照目的顶点进行分区，如此对分区依次进行访问，以后每次访问的数据总在当前数据访问位置的下方。

### 2.5.2 异步IO

异步IO的概念是相对于同步IO而言。在同步IO中，IO请求发出后，当前的执行过程会被阻塞直到IO操作完成返回结果之后，当前的执行过程才会继续执行。在大规模图处理过程中，会频繁的处理IO操作，同步IO会降低处理的效率。对于异步IO而言，异步调用过程发出之后，当前的执行过程会继续进行，并不会等待将消息写入磁盘完成，提高图计算的效率。

### 2.5.3 内存映射

内存映射是由操作系统提供的一种将文件映射到某一进程的地址空间的一种技术。该技术的主要目的就是为了提升对磁盘数据的访问效率。通过内存映射技术，进程或者应用程序就可以对文件进行随机访问。在32位的操作系统上，内存映射的大小有限，最大只能映射2GB的文件；在64位的操作系统中，通过内存映射技术可以映射非常巨大的文件。有学者[40,41]对内存映射在大数据处理上进行相关研究，事实证明内存映射不仅有助于快速处理数据同时还能节省大量额外

的IO管理等操作。所以，对于单机上的大规模图处理而言，内存映射技术可以很好的解决图处理过程中的随机访问的问题。

## 2.6 小结

本章首先对图相关的概念和术语进行了介绍，并对图的相关常用算法进行说明；然后针对大规模图处理分别从计算模型、并发技术以及磁盘IO三个方面进行了详细的讨论和说明：在计算模型中，以Vertex-Centric模型作为介绍的中心，并且详细的说明为什么在Vertex-Centric之外需要BSP计算模型，同时与Edge-Centric模型进行了对比；接下来讨论多线程并发模型在大规模图处理中的应用细节，同时指出存在于多线程并发技术中的复杂性，继而引入本文的并发技术，Actor角色并发模型，并对本文选择的Actor并发库Kilim进行了简要的介绍；最后，从磁盘IO的角度对大规模图的处理进行相关的介绍与分析。

# 第3章　GPSA系统设计与实现

## 3.1　计算模型简介

大规模的图处理根据不同的计算平台会展现出不同的特性。在分布式或者云平台上，大规模图被分割分配在不同的计算节点上，主要表现出计算密集型的特点。而单机的计算平台上，大规模图系统则同时表现出计算密集型和IO密集型两个特点。因此，在单机系统中设计大规模图处理系统的时候就需要同时兼顾两个特性。

由于大规模图的处理无法满足程序局部性的特征，很容易引起数据的随机访问，高效地处理大规模图变得非常困难。因此，针对大规模图处理有相关研究提出了全新的计算模型来适应大规模图的数据随机访问的特性。目前，已经存在几种比较成熟的计算模型。

### 3.1.1 BSP计算模型

在大规模图处理系统中，BSP模型的实现由可以进一步分为两大类：以顶点为中心和以边为中心。在以顶点为中心的计算模型中，由用户提供针对每个顶点的处理函数，顶点之间互相通信。整个计算过程由一系列的超级步组成，在每一个超级步内，一切处理围绕顶点展开，由顶点完成一系列同步计算。首先，顶点处理来自入边的更新消息，完成本超级步内的计算。然后，再将自己的更新消息发送给其邻接顶点。最后，所有顶点完成本超级步的处理之后，整个图进入下一个超级步。X-Stream以边为中心的计算模型则是从边的视角出发，将计算组织成一系列的迭代过程，将计算过程分为发散和收集两个步骤。在发散阶段，X-Stream通过边将更新信息由源点发送到目的顶点，在收集阶段，则对边的目的顶点进行更新。然后，对所有边进行迭代循环处理，当所有边都完成处理之后，进入下一个超级步。

### 3.1.2 GraphChi异步计算模型

与BSP的同步计算不同的是GraphChi默认最新的消息对于后续顶点总是可见。GraphChi的异步实现主要基于磁盘的PSW(平行滑动窗口)。GraphChi解决了随机访问所带来的问题，将图分为多个shard和interval，其中interval表示处理顶点的区间，shard包含了目标顶点在当前interval的边，并且这些边按源顶点顺序保存。在GraphChi处理某一个interval时，异步的处理模型就是当前的interval的顶点的状态的更新对于后面进行处理的interval内的顶点而言是可见的。在这样的情况下，

将会存在消息被覆盖的情况出现。

## 3.2 BSP计算模型改进

BSP模型在分布式图处理系统中被广泛采用，主要是因为BSP模型非常适合分布式的图计算环境。因此，很多基于分布式内存的大规模图处理统都构建在BSP模型的基础上，例如Pregel、GPS等。

鉴于当前的计算机以多核为主，单个计算机的计算能力、并发处理能力已经有了很大的提高，将BSP模型从分布式的环境中迁移到这样的多核的计算机上不仅可以充分利用多核的计算资源，还能够降低图计算的处理成本。同时，考虑到单机图处理系统与分布式图处理系统的兼容性，本文首先对BSP模型中存在的问题进行了分析，然后针对这些问题提出New BSP模型，并在新BSP模型的基础上实现一个便捷、高效、可靠的单机图处理系统GPSA。

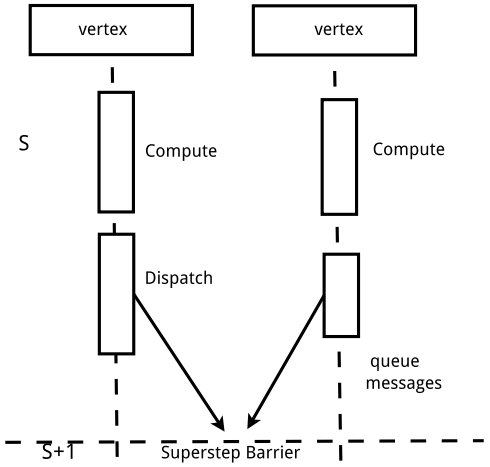



图 3.1 传统BSP模型

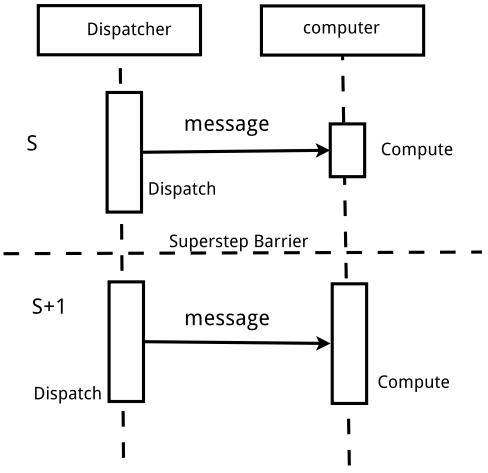



图 3.2 改进后的BSP模型

### 3.2.1 传统BSP模型的缺陷

由于BSP模型的同步路障机制的存在，那么在整个BSP模型的垂直方向上，影响最终效率的关键就是最晚完成的任务，如图3.1所示。在基于传统BSP计算模型的图处理系统中，如图3.1所示，图的处理主要分为两个阶段：以顶点为中心的计算(Compute)过程和消息的分发(Dispatch)过程。这两个过程按照严格的串行方式执行的，消息分发过程对计算过程存在数据依赖，形成强耦合。这种串行执行的方式，进一步延长了垂直方向上的任务完成的时间，降低了执行效率。在以顶点为中心的图计算中，消息是顶点之间的主要数据交换，顶点调用用户提供的计算函数对数据进行处理。消息包含计算所需的全部数据，所以相对于计算过程而言消息的具体分发过程是可以透明的。由于消息的生成和处理分布在两个相邻的超级步中，在该超级步结束之前，需要缓存大量的消息，并且在下一个超级步开始之前，这些消息不会被处理掉，增加额外的IO开销。目前，基于BSP计算模型的

图处理系统多任务并行处理的方法主要是采用多线程的技术。而线程是操作系统的基本调度单位，在操作系统中利用多线程在并发度上就会受到很大的限制。另外，由于需要保存大量的数据，当线程处理IO操作的时候，线程的调度会引发上下文的频繁切换，会对处理效率造成影响。

### 3.2.2 Actor-BSP模型

消息的分发过程和计算过程之间的主要依赖关系是消息，那么消息分发过程是消息的生产者，计算过程则是消息的消费者，两者之间以生产者和消费者的模式进行共存，从而将两者从严格串行的模式中解耦出来，如图3.2所示。在改进后的BSP模型中，消息分发过程和计算过程位于两个单独的执行流程中。消息分发过程主要负责消息的生成和转发。计算过程侦听消息，当消息达到，计算过程则负责对消息进行处理及数据更新。在语义上计算过程和消息分发过程是相互独立的轻量级调度单位。现有的BSP的实现在语义封装上往往是以顶点为中心，线程负责处理顶点。所以，在实现New BSP的模型中，采用轻量级、能够异步处理消息的并发模型成为关键。Actor并发模型相比于线程而言更加轻量级，有着更好的并发度。同时，Actor在语义上与Vertex的语义较为接近，兼顾了线程的调度执行和以顶点为中心的特点。

## 3.3 数据组织

单机的图处理系统同时具有计算密集型和IO密集型两个特点，合理的数据组织方式对整个系统的性能具有重要的影响。本节首先分析在Actor-BSP模型中数据的访问行为，然后根据数据的行为对GPSA的数据的组织方式进行说明和解释。

### 3.3.1 数据访问行为

在Actor-BSP模型中，Actor替换顶点成为整个计算展开的中心对象，之前顶点之间的消息的传递转变为Actor对象之间的消息通信，从而导致新模型在数据读写方式上与传统的BSP模型不同。本小节从消息、顶点以及边的角度出发结合Actor-BSP模型展示其数据读写方式的行为特征。

首先，消息无需缓存。由于Actor-BSP模型中，由于生产者和消费者的组合，计算Actor监听消息到达事件，一旦事件到达消息就可以被及时处理。新模型无需为下一个超级步保存大量的消息，减少消息缓存的IO操作。

其次，只有分发Actor才会访问边。由于分发Actor主要任务就是生产消息，而消息的产生和发送都与边有着紧密的联系。但是，计算Actor不关心边的状态，只关心消息到达的事件以及如何处理消息与更新数据。

再次，顶点状态数据信息需要常驻内存来支持随机访问。在以顶点为中心的模型中，顶点是有状态信息的，当顶点开始对消息进行处理的时候，顶点自身的状态会包含在自己的数据域，并且参与到计算中。无状态的Actor不会持有特定消息所有者的顶点状态信息。如果计算Actor接收到消息，它需要获取消息所有者的状态信息，然后调用用户提供的函数对消息进行处理。然而无状态的Actor并不记录消息到来的顺序及消息所有者的任何信息，所以无状态的Actor在处理消息的时候需要能够随机获取消息所有者的状态信息。

最后，双份状态信息同时参与计算。在传统模型中，顶点的状态信息需要额外保存一份来判断状态是否发生更新。而在Actor-BSP模型中，顶点的状态信息的双份保存主要是因为分发Actor和计算Actor对状态信息读取的目的不同。分发Actor获取顶点的状态信息主要是产生消息，而计算Actor获取状态信息主要是用于处理消息，并且替换更新的状态信息。所以两者在获取的状态信息是不一致的。分发Actor获取的状态信息主要是来自初始化或者上一个超级步的结果，而计算Actor获取的状态信息却总是当前超步的最新结果。

### 3.3.2 磁盘IO

GraphChi和X-stream需要将大量的数据写入到磁盘中，因此对提高IO的性能有着非常高的要求。所以，Graphchi不遗余力的设计平行滑动窗口(PSW)，并藉此实现异步的计算模型，并且为了实现PSW,GraphChi需要做非常多的预处理来满足要求。而X- stream也是从避免随机访问的角度出发采用顺序访问的Scatter和Gather两个阶段对图进行处理，同时采用异步IO的方式来获得更好的性能。

然而，在新的计算模型中，数据的访问方式发生较大的变化。不需要缓存消息意味着不需要为缓存大量的中间消息设计复杂的磁盘IO功能。边与顶点的分离，使得要处理的数据量减少很多。相比较整个庞大的图而言，顶点的状态信息就显得适合常驻内存进行处理来获取更好的性能。由于不需要向磁盘写入大量的数据，出于容错性的考虑，我们采用内存映射的方式对顶点的状态信息进行组织。而对于庞大数量的边而言，松散的组织结构可以省去一些耗时较长的预处理，同时满足顺序访问的特点，避免复杂的IO优化操作。

### 3.3.3 数据组织设计

图的数据主要分为两个部分，顶点的状态信息和边。其中，顶点的状态信息以二进制的格式存储在内存映射文件当中。边则以行压缩（Compressed Sparse Row）格式保存在磁盘上。

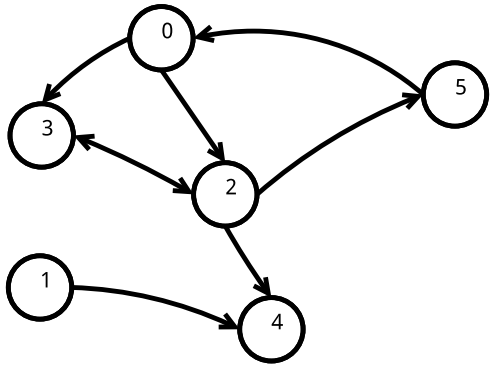



图 3.3 例图

```
vertex 0: 0x80000001  0x80000001
vertex 1: 0x80000002  0x80000002
vertex 2: 0x80000003  0x80000003
vertex 3: 0x80000004  0x80000004
vertex 4: 0x80000005  0x80000005
vertex 5: 0x80000006  0x80000006
```

图 3.4 两列存储

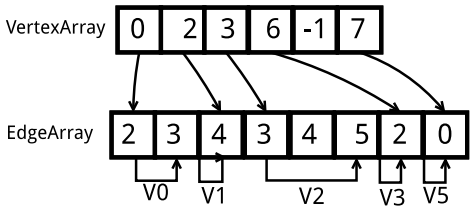



图 3.5 CSR存储

(a) 2,3,-1,4,-1,3,4,5,-1,2,-1,-1,0,-1

(b) 2,2,3,-1,1,4,-1,3,3,4,5,-1,1,2,-1,-1,1,0,-1

图 3.6 顺序存储

由于顶点的状态信息需要保存两份以供两种actor访问，所以在顶点状态信息的内存映射文件中，同一个顶点的状态信息以相邻的方式保存两份，就如同两排并列的数据，如图3.4所示。通过这种简单的方式，可以通过顶点*id*计算顶点在整个文件中的偏移值来快速的获取顶点$V$的状态信息的值，即$|V| * sizeof(Val) * 2$。当该内存映射文件被映射到内存中后，整个访问的过程就像在操作一个数组一样方便。

CSR格式将顶点*id*和边分别用两个数组进行保存，其中边数组保存边的目的顶点*id*并按照出发顶点*id*排序，而在顶点数组中顶点的*id*就是数组的索引，每个顶点保存第一条边在在边数组中的索引。此时，如果需要遍历第*i*个顶点的边，那么只需要访问EdgeArray[VertexArray[i]]，EdgeArray[VertexArray[i]+1]，…，EdgeArray[VertexArray[i+1]]就可以完成该顶点边的遍历。如图3.5所示，在遍历顶点*Vertex 2*时，就可以通过访问[EdgeArray[3],EdgeArray[6])对边顶点进行遍历。CSR格式的空间效率是O(n+m),其中n和m分别是边和顶点的个数。另外，还可以在CSR格式的基础上采用顺序存储的方式，每个不同顶点的边之间用分割符加以分别，如图3.6(a)所示，不同顶点之间可以用*-1*作为分隔符，对边的遍历只需要读取到-1为止即可，除此之外还可以保存除边之外的其他信息，例如出度，权值等，在图3.6(b)中就额外保存了每个顶点的出度，数组中第一个索引位置的值是2，表示此处顶点的的出度是2，紧接着的是两条边，当读取到-1的时候，顶点的*id*递增加一。

## 3.4 消息分发

消息是本系统的重要关注和处理的对象，虽然Actor-BSP模型的优势使得无需为消息进行额外的缓存，但是合理的消息产生和分发策略对系统的效率有着重要的影响。

一般来说消息的主要有两部分内容组成：目的顶点和消息的值。目的顶点表明该消息前往的顶点，同时也会影响由哪个具体的计算Actor来处理该消息。消息的值则是计算Actor用来进行计算的数据。虽然消息的生成工作主要是由分发Actor负责，但是消息的值的生成方法根据应用的不同具体的生成算法也不一样。例如，在遍历应用中，消息的值则是一个表示从源点开始到目前顶点的一个层次数，而在PageRank算法中，消息的值则是当前顶点的权值均分到他的出边邻接顶点的浮点数。所以，具体的消息的实现过程，由用户自己实现，分发Actor则调用具体的生成算法产生消息。

当消息生成之后，分发Actor需要将消息分发给计算Actor。在Actor-BSP模型中，以顶点为中心的方式被无状态的Actor代替，计算Actor需要均衡的处理这些消息。虽然Actor的上下文切换远比线程轻量级，但是Actor的调度依然可能成为性能的制约因素，所以需要对消息的分发过程进行额外的硬性控制来避免某个计算Actor的负载过重。理论上来说，每个计算Actor都应该处理差不多个数的消息，但是由于图结构是无规律的，消息的发送目的地也有很大程度的随意性，在短时间内平均地将消息发送给计算Actor的可能性也微乎其微。为达到这样的目的，GPSA在预处理阶段对计算Actor进行了任务分配，将消息按照目的顶点的次序均分给计算Actor。在分发的过程中，如果某个计算Actor的邮箱已满，无法接受更多消息，那么该消息将会发送到目前消息数目最少的计算Actor的邮箱中，从而实现动态均衡。

## 3.5 数据更新

计算Actor接收到消息之后，调用用户实现的具体计算函数，如果经过计算之后顶点的状态发生改变，那么计算Actor就需要更新顶点的状态信息写入内存。由于计算Actor和分发Actor对数据访问的行为不同，所以需要两份不同的顶点信息的拷贝。其中一份拷贝供分发Actor查询使用，该数据是上一个超级步的结果；另外一份拷贝则供计算Actor进行更新。由于两份数据以“Two-Column”的方式进行保存，两份数据就可以以列进行区分，如图3.7所示。计算Actor和分发Actor对这两列数据依次交替访问，计算Actor访问的数据列是计算列，而分发Actor访问的数据列是分发列。

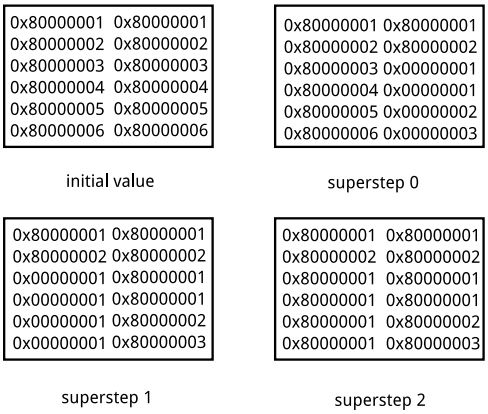



**图 3.7 数据更新**

在初始化结束之后，顶点的两列状态信息是相同的。分发Actor访问分发列的数据，然后计算Actor对计算列数据进行更新。在计算的过程中，为了避免消息的重复发送，同时保证计算Actor能够取得正确的顶点状态数据，将状态数据的最高位作为标识位。如果状态数据的最高位为1，则表示该数据在上一个超级步中没有发生更新，分发Actor将会跳过该顶点；若高位为0，则说明数据在上一个超级步中发生更新，分发Actor需要将该更新发送出去。该约束条件在以下两个情况下会被忽略：

- 超级步为0的情况：在第0个超级步中，顶点的状态信息最高位被初始化为1的，所以在第0个超级步中分发Actor不需要判定高位的情况。
- 错误恢复的情况：如果GPSA在某次运行中崩溃，需要从某一超级步恢复，此时恢复的时候分发Actor无需判定高位情况。

那么，顶点的状态信息的更新由分发Actor和计算Actor协同完成，分发Actor的主要更新步骤如下：

- 如果当前超级步为0或者是从某一个超级步恢复，分发Actor依次读取数据并将状态信息发送出去。
- 如果当前超级步大于0并且状态信息的高位若为1，则表明该数据尚未被更新，分发Actor将会跳过该数据。
- 如果当前超级步大于0并且状态信息的高位若为0，分发Actor对该数据的更新信息进行分发，并在消息分发完成后，将该数据的高位置1。

计算Actor的主要更新步骤如下：

- 如果超级步为0，计算Actor从计算列读取顶点信息，无需判定高位情况。
- 如果超级步大于0，计算Actor如果读取到的数据高位为1，则表明该数据不可用，此时计算Actor将会读取该顶点上一个超级步后的最新结果，即从分发列对应位置获得最新值。
- 如果超级步大于0，计算Actor如果读取到的数据高位为0，则表明当前位置

的数据是最新的，将会直接读取数据进行计算。

- 无论哪种情况，如果计算之后顶点状态信息发生更新则把计算之后的新数据写入到当前顶点的计算列，否则不更新，下次读取数据依然从分发列读取数据。

图3.7是图3.3中的图求联通分量时，顶点的状态信息的更新流程。在初始化时，每个顶点的状态信息依次被初始化为它自身的$id+1$，两列数据相同并且最高位为1。

在超级步*0*中，分发Actor不判定顶点状态信息的高位情况，将所有顶点的状态信息发送出去，计算Actor在接收到消息之后根据消息值与顶点的值比较，选择较小的值作为新的顶点状态信息的值，如顶点2在超级步0中接收到来自顶点0的消息值为1，接收到来自顶点3的消息值为4，则计算Actor计算之后将顶点2的状态信息更新为1，并且高位为0，而顶点0由于在超级步0中，并没有发生更新所以结果为1，并且高位为1。

然后，超级步*1*中，计算列和分发列互换位置，可以看到顶点2、3、4、5发生更新所以将会发送消息并在顶点消息发送完之后将相应的高位置1，而顶点0、1高位为1，将会被跳过。依次类推，直到计算完成。

## 3.6 GPSA工作模块实现

GPSA的实现基于JAVA和Kilim角色并发框架。在GPSA中，输入是在上文中详述的以二进制形式保存的数据文件，包括顶点的状态信息和边的信息；输出则是经过计算之后图中顶点的状态信息。如图3.8，GPSA主要包括三个功能模块：预处理、Manager管理模块、Actor工作模块。预处理模块主要涉及对计算开始之前的初始化工作；Manager管理模块负责对计算进行协调与控制；分发Actor负责消息的产生和分发；计算Actor则是计算的基本执行单元。

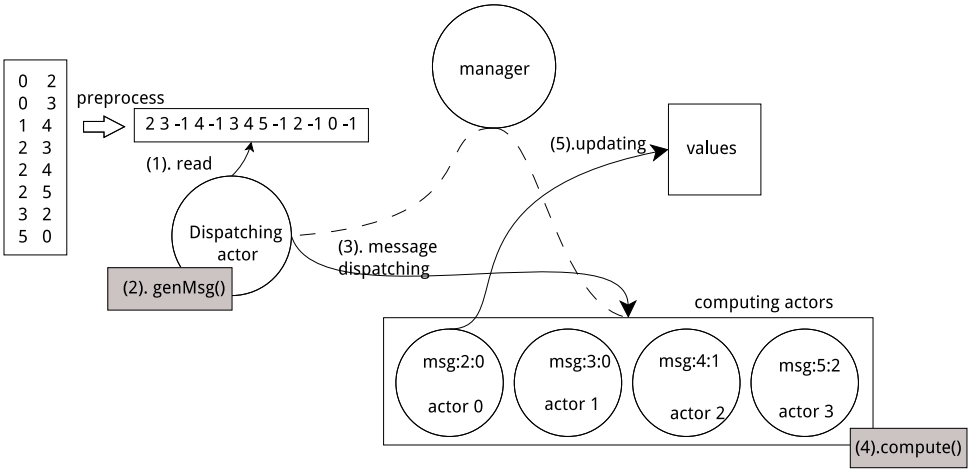



图 3.8 系统架构

### 3.6.1 预处理

GPSA默认对边集存储的数据进行预处理。首先，边按照起始顶点的大小升序排序。虽然GPSA的新模型对分发过程和计算过程进行了分离，GPSA理论上只要能够获取边、起始顶点的顶点状态信息和用户自定义的消息生成的函数就可以发送消息并驱动计算Actor进行计算，也就是说GPSA可以处理原始的以边集存储的图。但是GPSA的新模型仍然是基于BSP模型的，耗费时间最长的任务将决定实际的计算效率，而直接对原始的边集数据进行计算，容易造成任务分配不均，计算效率低的结果。另外，在新模型中图的顶点信息是需要进行初始化并且映射到内存中的，需要与边区别对待。在GPSA中，图是以CSR的格式进行存储的，需要将顶点和边分开，计算出图的一些基本信息，例如，边数、顶点数等，同时对任务进行分割。首先，预处理模块读取边，对边按照起始顶点的大小按升序的方式进行排序，如果起始顶点相同则按照目的顶点的大小升序排序。接下来，对排序之后的边进行CSR格式转换，将转换后的CSR格式的图数据写入磁盘，同时计算出图的基本信息，例如顶点的出度和入度、顶点个数，边的个数等等。

任务分割主要指计算任务的分割。对于消息分发Actor而言，顺序读取CSR数据即可；对于计算Actor而言，每个计算Actor所处理的消息个数应该相差不大，同时为避免有两个Actor同时操作同一顶点信息的更新而引起同步写，所以GPSA采用了一种“区间”的处理方法，即每个计算Actor负责连续的顶点闭区间$[i, j]$，每个区间接受的消息总数目接近每个计算Actor负责消息的平均值。

假设图的总边数为$m$，计算Actor个数为$n$，则每个计算Actor的区间处理的消息数目为$k$满足以下条件：

$$\frac{m}{n} \leq k < \frac{m}{n-1}$$

预处理模块根据顶点的入度计算每个Actor所应处理的消息个数，并生成相应的区间对象，将其传递给Manager管理者。

### 3.6.2 Manager管理模块

Manager主要负责初始化计算，协调分发和计算的有序进行以及异常处理。在计算开始执行之前，图数据需要准备就绪，Manager调用预处理模块对图进行处理，根据图的基本信息计算出最小的顶点*id*和最大的顶点*id*，调用用户定义初始化函数对顶点的状态信息进行初始化，并将结果写入顶点信息内存映射文件，完成对顶点初始信息的初始化。另外，manager需要根据用户指定的分发Actor和计算Actor数目，初始化对应数目的Actor。

初始化工作完成之后，Manager主要对计算过程进行协调。如算法3.6.1所示，首先Manager向分发Actor发送*ITERATION_START*信号。接收到*ITERATION_START*

**算法 3.6.1** Manager Execution Loop

```
Input:
    Initialize dispatchers,computers,mailbox,
 1: counter ← 0
 2: currIte ← 0
 3: while currIte < endIte do
 4:   for dispatcher ∈ dispatchers do
 5:     dispatcher ← ITERATION_START
 6:   end for
 7:   while (signal ← mailbox.get()) == DISPATCH_OVER do
 8:     counter ← counter + 1
 9:     if counter == dispatchers.length then
10:       counter ← 0
11:       break
12:     end if
13:   end while
14:   for worker ∈ computers do
15:     worker ← COMPUTE_OVER
16:   end for
17:   while (signal ← mailbox.get()) == COMPUTE_OVER do
18:     counter ← counter + 1
19:     if counter == computers.length then
20:       counter ← 0
21:       break
22:     end if
23:   end while
24:   currIte ← currIte + 1
25: end while
```

信号的分发Actor会开始访问存储在磁盘上的CSR格式的图数据，调用用户定义的消息生成规则函数打包消息。当分发Actor在当前超级步的消息分发任务完成之后，分发Actor用*DISPATCH_OVER*信号通知Manager。如果所有的分发Actor都完成之后，Manager紧接着向所有的计算Actor发送*COMPUTE_OVER*信号。如果计算Actor处理到该消息则会意识到本超级步的计算工作已经完成，此时它会回复Manager一个同样的*COMPUTE_OVER*。最后，如果计算满足结束条件，Manager向所有的Actor发送*SYSTEM_OVER*，当Actor接受到该信号将会退出自身的执行循环，结束生命周期。

### 3.6.3 Actor工作模块

Actor模块是系统中最重要的组成部分，该模块主要负责消息的生成和计算并更新数据。在该模块中主要包含两种无状态Actor：分发Actor和计算Actor。

分发Actor的工作流程如算法3.6.2所示，分发Actor等待来自Manager的*ITERATION_START*的信号，接收到信号后分发Actor开始进入执行循环，检查处理的边的起止信息，从起止信息的起始位置开始顺序从磁盘上读取一条边，获取该边起始顶点的顶点的状态信息，如果当前超级步不是0并且信息高位为1，则表明当前顶点的状态信息在上一个超级步中没有发生更新，需要跳过

**算法 3.6.2** Dispatcher Execution Loop

```
1:  signal ← mailbox.get()
2:  while signal! = SYSTEM_OVER do
3:    if interval! = null then
4:      reset()
5:      while curoff < endoff do
6:        val ← getValue(sequence)
7:        if isHighestBit(val) == 1&&currIte! = 0 then
8:          skip(sequence)
9:        end if
10:       if isHighestBit(val) == 0|||currIte == 0 then
11:         vid ← readEdge(curoff)
12:         while curoff < enfoff do
13:           if vid == −1 then
14:             break
15:           end if
16:           msg ← genMsg()
17:           DispatchStrategy(vid, msg)
18:         end while
19:         setHighesetBitTo1()
20:       end if
21:     end while
22:   end if
23:   notifyManager(DISPATCH_OVER)
24:   signal ← mailbox.get()
25: end while
```

该顶点，继续处理下一个顶点，*skip(sequence)*函数，则会让当前的顶点*id*加1，同时跳过该顶点的所有边，寻址到下一个顶点的第一条边的偏移地址。若当前超级步为0或者高位为0，则表明该顶点处于初始化之后的状态或者在上一个超级步中发生了更新，分发Actor调用用户提供的*genMsg()*实现生成消息，根据分发策略将消息发送出去。如果读取边的目的顶点为-1，则表明当前顶点处理完毕，并将该顶点的状态信息状态最高位置1表明处理完成。如此周而复始，直至完成分发任务，分发Actor用*DISPATCH_OVER*通知Manager。

**算法 3.6.3** Compute Execution Loop

```
1:  msg ← mailbox.get()
2:  while signal! = SYSTEM_OVER do
3:    if msg == COMPUTE_OVER then
4:      notifyManager(COMPUTE_OVER)
5:    else
6:      to ← msg.dest()
7:      msgVal ← msg.val()
8:      val ← getVal(to)
9:      newVal ← compute(val, msgVal)
10:     if newVal! = val then
11:       update()
12:     end if
13:   end if
14:   msg ← mailbox.get()
15: end while
```

计算Actor的工作流程如算法3.6.3所示，计算Actor等待来自Manager的消息，然后进入执行循环体。否则计算Actor从消息中提取目的顶点和消息的值，根据目的顶点从顶点的状态信息内存映射文件中读取顶点值，然后调用用户提供的*compute(val,msgVal)*实现对其进行计算，计算返回一个新的顶点状态信息。如果新的顶点状态与之前的值相比发生了改变，则将新数据更新到内存映射文件，否则丢弃。如此迭代，直到计算Actor接受到的信号是*COMPUTE_OVER*，则表示当前的超级步的计算完成，同时用*COMPUTE_OVER*回复Manager。如果计算Actor接收到*SYSTEM_OVER*信号，则会结束计算退出循环。

## 3.7 小结

本章首先介绍常见的用于图计算的模型，并讨论了BSP模型中的缺陷。接下来在传统BSP模型的基础上，使用Actor将计算过程和分发过程分开。然后围绕Actor-BSP模型中的数据行为、磁盘IO、消息分发和数据更新等内容对GPSA系统进行设计和规划。最后，在此基础上，本章对GPSA的系统实现进行阐述和说明。从功能模块上划分，GPSA主要分为预处理模块、Manger管理模块、计算Actor和分发Actor模块。预处理模块对数据进行数据格式的转换与信息分离，以及任务分配等工作。Manager则是系统的执行计算的控制单元，负责协调计算Actor和分发Actor的开始和进行，以及一些异常处理。计算Actor和分发Actor模块则是主要的运算模块，其中分发Actor从磁盘读取一条边，根据用户的消息生成规则产生消息，并将消息分发给计算Actor，而计算Actor在接受到消息之后，会根据消息访问顶点信息，调用用户的*compute(val,msgVal)*函数进行计算并取得返回值，然后判定是否发生更新，并将更新数据写入内存。

# 第4章　GPSA系统分析

本章对GPSA系统进行高效性、可行性与稳定性分析。首先从Actor-BSP模型入手解释GPSA的正确性和高效性，然后从IO角度对GPSA在单机系统上的可行性进行说明，接着从容错性的角度介绍GPSA的稳定性。最后，通过如何在GPSA系统上实现常用的图应用进行举例说明。

## 4.1　Actor-BSP模型分析

首先，Actor-BSP模型与以往的以顶点为中心的模型的异同使得Actor-BSP模型具有类似的语义，从而保证模型的正确性。Actor-BSP模型是以Actor角色作为中心的，但是新模型并没有完全消除以顶点为中心的概念。而仅仅是对其进行淡化，使得在顶点上的计算转移到角色上进行，同时能够兼顾顶点状态的更新而无需关注顶点和角色之间的强制的依赖关系。另外，Actor角色还分担了传统意义上线程的执行者的角色。以往线程负责对顶点进行遍历并调用顶点上的消息处理函数，在新模型中则是Actor角色负责遍历顶点并调用提供给Actor的消息处理函数。

其次，Actor-BSP模型将计算过程和消息分发过程分离使得GPSA能够获得性能提升。在以顶点为中心的图计算中，对于顶点而言，其计算过程和消息发送过程是紧密相关，不经过对顶点的状态信息进行计算就不能发送顶点的更新消息。所以，此时顶点的计算过程和消息发送过程是严格意义上的顺序执行。对于BSP的应用而言，执行时间最长的任务决定整体的效率。Actor-BSP模型是在执过程上完全异步计算的模型，计算过程与消息分发过程相分离，使得两个过程可以在一定程度上并行执行。在Actor-BSP模型中，Actor分为两种类型：负责消息分发过程的分发Actor和负责计算过程的计算Actor。得益于分发Actor和计算Actor之间的松耦合设计，两者之间的映射关系变得相当的灵活，缩短BSP模型在垂直方向上的执行时间，提高处理效率。

最后，Actor-BSP模型的轻量级与高并发。Actor并发模型本身就是一种轻量级的高并发的编程模型。Actor擅长处理消息，不同的Actor之间通过邮箱通讯，摒弃基于内存共享并发模型的同步和锁等概念。

其次，分发Actor 和计算Actor 组成了生产者和消费者模型，当消息到达计算Actor之后，该actor会被调度执行，处理消息，从而无需保存这些消息，避免将消息进行缓存的IO开销。

## 4.2 IO分析

单机系统上的图处理系统不仅仅需要进行大量的计算，与此同时还会涉及大量的IO操作。以往的图处理系统中，除了在更新顶点状态信息的时候需要写入大量的数据，还需要缓存大量的消息以供系统在下一个超级步进行计算。所以，整体而言写入磁盘的数据非常多，IO开销非常巨大。由于在单机系统中，大量的IO操作会成为系统的主要瓶颈，所以其他系统往往从优化IO问题着手，通过避免图数据的随机访问问题，批量写入大块数据来解决IO问题。

与其他单机图处理系统不同，GPSA没有刻意去避免随机访问的问题，GPSA首先对图数据在新模型下的访问行为进行详细的分析和讨论，然后根据访问行为的不同，将图数据分为两部分。一部分支持随机访问，通过内存映射的方式使之常驻内存；另外一部分则不支持随机访问，保存于磁盘通过顺序访问的方式进行数据的读取操作。其次，GPSA最大的IO优势之一：GPSA无需缓存大量的消息到磁盘中。在GPSA中，Actor分为两种不同的类型：分发Actor和计算Actor。分发Actor负责消息的产生和发送工作；计算Actor则负责消息的处理和顶点的状态信息更新。从Actor的角度分析，分发Actor 和计算Actor 组成了生产者和消费者模型，当消息到达计算Actor之后，受到消息驱动的计算过程便会开始执行从而无需保存这些消息，避免将消息进行缓存的IO巨大开销。

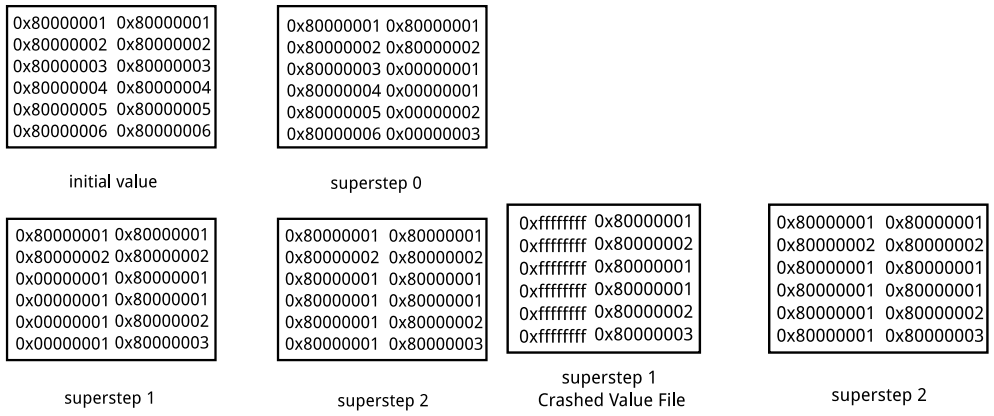
## 4.3 容错性分析

容错性是系统安全稳定运行的重要保证。在GraphChi和X-Stream中，实现容错性非常困难。这是因为它们需要保存大量的数据信息和图处理的中间结果。但是，系统崩溃往往是一刹那的事情，在短时间内完成大量数据的信息保存时非常困难的。

得益于GPSA将图数据的区分对待，容错性得到很好的保证。本系统中发生改变的数据只有顶点的状态信息，并且无需保存大量的消息，所以只要能够在系统崩溃时找到一个保存了完整图顶点状态信息的时间点就能从该时间点对系统进行恢复。在本系统的设计中，顶点的状态信息是以“Two-Column”的方式保存在内存映射文件中的，同时计算Actor和分发Actor分别访问不同数据列，并且分发Actor只能读取数据，而只有计算Actor才能写入数据。也就意味着，分发Actor所读取的数据列的结果就是上一个超级步的图顶点状态信息的完整备份。

如图4.1所示，如果系统在超级步1崩溃，此时顶点状态信息如图4.1 (b)所示。在超级步1，右边是分发列，左边是计算列。所以右边的分发列就是超级步0完成

之后的正确结果。系统可以从该列进行数据恢复操作。




(a) 正常时的执行情况　　(b) 崩溃时的执行情况

图 4.1 容错性

## 4.4 应用示例

在GPSA的框架上实现应用非常简单，一般情况下只需要实现*Handler*接口，在该接口必须要实现的函数有两个：init(sequence)和compute(val,msgVal,args)。其中，init函数主要用于初始化顶点的状态信息，而compute方法则用于在计算Actor处理消息时进行调用。除此之外，还有一些可选函数，例如消息生成函数、更新判定等，如果不实现这些可选函数，则会以默认方式进行计算。例如，若不实现消息生成函数，则默认将顶点的最新状态打包进消息中进行发送。

### 4.4.1 PageRank

PageRank是著名的网页排名算法。在GPSA中实现该算法，需要实现*Handler*接口，将顶点的初始值初始化为1.0，然后当计算Actor接收到消息进行计算时，需要对这些消息进行累加，如算法4.4.1所示。

**算法 4.4.1** PageRank

**Input:**

$val, msgVal, args$

**Output:**

$newVal$

1: **if** $args[0]$ **then**
2: **return** $0.15f$
3: **end if**
4: **return** $0.85 * msgVal + vertexVal$

### 4.4.2 BFS

BFS是对图进行广度遍历的算法。在GPSA中实现该算法则需要指定起始遍历的顶点，起始顶点的值初始化为0，其他顶点初始化为无穷大，通过发送消息并对消息进行计算可以得到某一个顶点在广度遍历中在整个图中的遍历层次。对于一个顶点而言，假如*V*所处的层次为n，那么它的出边的顶点的层次在没有其他干扰情况下（例如，这些顶点同时也是层次为n-1的顶点的出边目的顶点）是n+1。否则，可以通过比较他的目的顶点的层次与它的消息中的新层次，最小的那个即为其在广度遍历算法中的层次。由于无法保证消息的发送和接收顺序，所以在计算函数中通过比较消息的值的层次的大小，并将值最小的消息加1之后作为返回值，具体实现如算法4.4.2。

**算法 4.4.2** Breadth First Search

**Input:**

$val, msgVal, args$

**Output:**

$newVal$

```
1: if msgVal < val then
2:     return msgVal + 1
3: end if
4: return val
```

### 4.4.3 CC

连通分量的实现过程与广度搜索类似，但是在初始化的时候，将所有顶点的值初始化为其自身*id*，然后在计算函数中通过比较两者的*id*的大小，选择返回较小值。这样，计算完成之后，处在同一个连通分量中的顶点拥有相同的*id*，具体实现如算法4.4.3所示。

**算法 4.4.3** CC

**Input:**

$val, msgVal, args$

**Output:**

$newVal$

```
1: if msgVal < val then
2:     return msgVal
3: end if
4: return val
```

### 4.4.4 小结

本章是GPSA系统的分析部分，在前面几章对图处理技术和GPSA系统的设计与实现介绍的基础上，进一步对GPSA系统进行了多个方面的分析。主要从系统整体的可行性、效率、稳定性和便捷性对系统进行分析。首先，对GPSA的计算模型从模型语义、结构和并发三个角度进行分析，解释GPSA高效的原因；其次，对GPSA的将图分为两部分，从而实现两种不同IO操作方式进行说明，并且指出GPSA无需缓存大量消息；接下来，对GPSA的“Two-Column”存储方式的容错性进行介绍，证明GPSA系统的稳定性；最后，通过举例说明在GPSA系统上开发的便捷性。

# 第5章　系统测试与分析

为了证实GPSA系统的实际性能，本章对该系统进行了测试和分析。首先从硬件、软件以及用作测试的数据集三方面介绍了测试环境；接着通过PageRank、连通分量和广度搜索三个应用，测试GPSA在不同数据集上的运行性能，并将结果与GraphChi和X-Stream进行对比分析；最后，从对单机多核的利用率角度对系统进行测试和分析。

## 5.1　实验环境

### 5.1.1 硬件环境

本文的系统与测试都是运行在相同的计算机上。该计算机CPU为32核，主频1.8GHZ的Intel i7 cores, 16GB内存，1TB硬盘，转速7200rpm。

### 5.1.2 软件环境

本文的操作系统为Ubuntu 12.04LTS，JDK7，Kilim。另外，本文用来对比的单机图处理系统分别为GraphChi（0.2.6 C++）与X-Stream。

### 5.1.3 测试数据集

为了能够全面的了解GPSA的运行性能，所以我们选择四个大小不同的数据集：Google数据集、soc-pokec数据集、soc-liveJournal数据集以及twitter-2010数据集。四个数据集的具体情况如下表所示：

表 5.1 测试数据集大小

| Name | Nodes | Edges |
|---|---|---|
| google | 875,713 | 5,105,039 |
| soc-pokec | 1,632,803 | 30,622,564 |
| soc-liveJournal | 4,847,571 | 68,993,773 |
| twitter-2010 | 41,652,230 | 1,468,365,182 |

## 5.2　性能测试

本文不仅对GPSA系统在不同应用、不同数据集上的表现进行了测试，同时还横向与其他单机图处理系统进行对比。为保证测试的公平与直观，GPSA、GraphChi和X-Stream均以默认设置运行，通过计算运行相同步骤所消耗的时间作

为性能对比的主要指标。所有的测试都是在相同环境进行，并且分别运行三次，最终结果是三次运行所消耗时间的平均值。由于GPSA和GraphChi在运行时需要指定运行的迭代次数，而X-Stream则不需要指定该迭代次数。考虑到图计算过程在最开始的若干超级步中的处理速度可以较为直观的反映出计算的效率，因此本文测试了开始前5个超级步的平均消耗时间作为对比的标准。另外，GraphChi和X-Stream的三个测试应用均采用其官方提供的实现。

### 5.2.1 google数据集测试

图5.1展示了三个系统的PageRank、Connect Component和BFS三个不同的应用在Google测试数据上的运行结果。通过对比可以发现，在PageRank和BFS应用中，GraphChi的性能最好，X-stream次之。在CC应用中，X-Stream的性能最好，GraphChi次之。在PageRank中，GPSA比GraphChi和X-Stream慢大约4倍；在BFS中，GPSA比GraphChi和X-Stream慢大约0.2倍；虽然在CC中，GPSA与GraphChi几乎持平，但是依然比X-Stream慢。

首先，google的数据集相对较小，整个图的文件大小在100MB范围内，可以完全载入内存。GraphChi和X-Stream省去了较多磁盘IO操作，在内存中可以完成所有数据操作。而GPSA由于将图的数据部分保存在磁盘中，所以会有部分的磁盘IO顺序读取。其次，框架的实现有区别。GraphChi和X-Stream使用C/C++实现，而GPSA使用JAVA和Kilim实现，会无可避免的出现垃圾回收等问题，所以在语言效率上，GPSA略逊一筹。

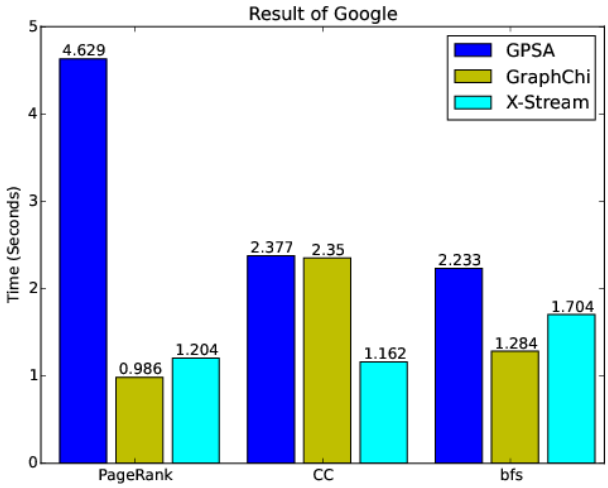



图 5.1 Google测试结果

### 5.2.2 soc-数据集测试

图5.2展示了GPSA、GraphChi和X-Stream在soc-Pokec数据集上分别运行PageRank、Connect Component和BFS三个应用的运行结果。通过对比

实验结果可以发现，GPSA的性能最好，GraphChi次之。在PageRank中，GPSA比GraphChi快大约0.3倍，比X-Stream快8倍；在CC中，GPSA比GraphChi快大约4倍，比X-Stream快约6倍；在BFS中，GPSA与GraphChi几乎持平，而X-Stream较差。

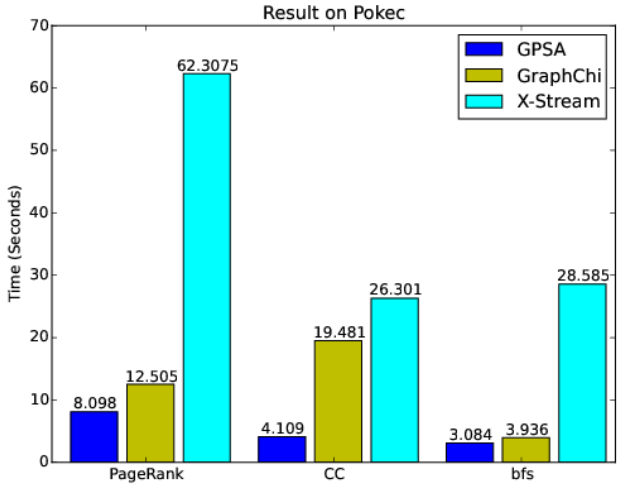



图 5.2 soc-Pokec测试结果

图5.3展示了GPSA、GraphChi和X-Stream在soc-liveJournal数据集上分别运行PageRank、Connect Component和BFS三个应用的运行结果。通过对比实验结果可以发现，GPSA的性能最好，GraphChi次之。在PageRank中，GPSA比GraphChi快大约0.3倍，比X-Stream快10倍；在CC中，GPSA比GraphChi快大约4倍，比X-Stream快约6倍；在BFS中，GPSA与GraphChi几乎持平，而X-Stream较差。

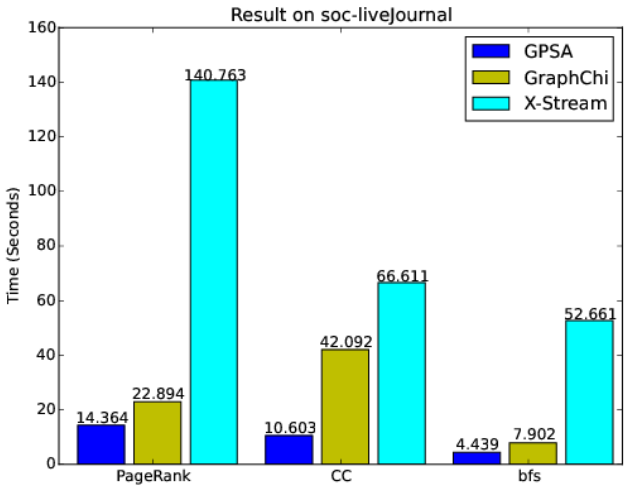



图 5.3 soc-liveJournal测试结果

通过5.2和5.3的实验结果可以发现实验结果具有一定的一致性，GPSA的测试性能与在Google数据上有着巨大的差别。首先，从上文数据集的信息介绍中可以发现soc-Pokec和soc-liveJournal两个数据集都是中等规模大小的图数据，无论是顶

点数目还是边的数目都比比Google数据集大的多。因此，GraphChi和X-Stream需要对数据进行分批载入，同时在超级步完成之后，需要写大量的数据到磁盘，引发大量的磁盘IO操作。而GPSA由于无需保存大量的数据到磁盘，所以在超级步更替的时候，无需额外的IO操作，效率较之两者有着明显提高。其次，可以看到GPSA和GraphChi的性能比X-Stream好。这主要是因为X-Stream是以边为中心的模型，所以在每次计算时都需要对所有的边进行迭代，而GraphChi以顶点为中心，在每次计算过程中，对于未发生过更新的顶点会跳过，减少消息量与计算量。虽然在GPSA中将顶点以Actor进行了替换，但是GPSA依然继承了对顶点状态的检查操作，使得与GraphChi有着同样的优势。

### 5.2.3 twitter-2010数据集测试

图5.4展示了GPSA、GraphChi和X-Stream在soc-Twitter-2010数据集上分别运行PageRank、Connect Component和BFS三个应用的运行结果。通过对比实验结果可以发现，在PageRank中，GPSA比GraphChi快大约2倍，比X-Stream快8倍；在CC中，GPSA比GraphChi快大约5倍，比X-Stream快约4倍；在BFS中，GPSA比X-Stream快约6倍。（由于GraphChi的官方提供的BFS实现在对数据进行预处理完之后，无法继续运行，所以此处的数据无从得知。）

Twitter2010是一个非常大的数据集。通过对该数据集的测试，可以充分展示GPSA在性能方面的提升与应对大规模图的计算能力。首先，在设计之初，GPSA将计算过程和分发过程进行分离，以一种重叠的方式并发运行，缩短了BSP模型中在垂直方向上的平均执行时间。其次，GPSA 将图的数据分为易变的顶点信息和不易变的边信息区别对待，部分保存在磁盘上，部分保存在内存中，实现了数据的随机访问，降低IO开销。最后，GPSA中计算Actor和分发Actor的协同工作使得无需保存大量的中间消息，只须关注计算结果，简化了计算过程。

## 5.3 多核利用率测试

GPSA是为运行在单机多核的系统上而设计的，所以对GPSA进行多核利用率的测试可以直观的了解GPSA。如图5.5到5.7所示，比较了GPSA、GraphChi和X-Stream在不同的数据集上，不同的应用上对多核CPU的利用情况。

通过测试结果可以直观的发现，X-Stream对多核CPU的利用率最高，GPSA次之，GraphChi则相对较差。GPSA 与GraphChi具有较好的伸缩性与灵活度，对CPU的利用率能够随着应用和数据的不同而灵活的调节,而X-Stream的灵活性和扩展性表现较差,如图5.6所示，X-Stream在soc-pokec与twitter两个数据集上都表现出了非常高的CPU利用率，可是相比于twtitter数据集的庞大，soc-pokec数据集的

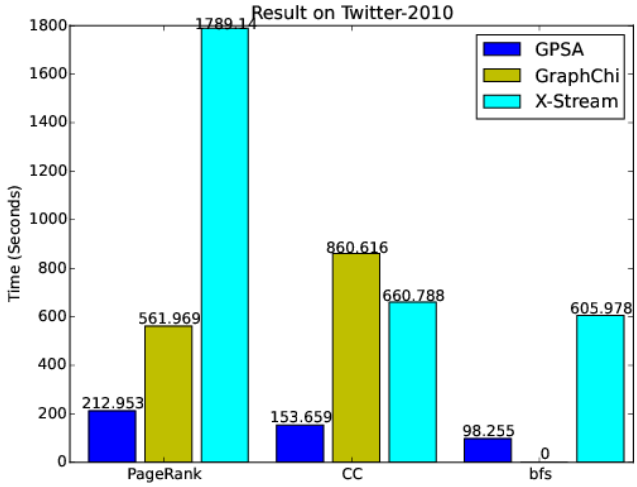



图 5.4 Twitter测试结果

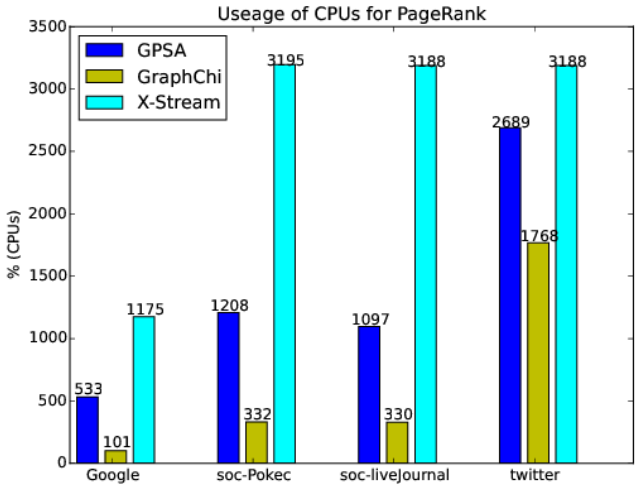



图 5.5 PageRank算法的多核CPU利用率

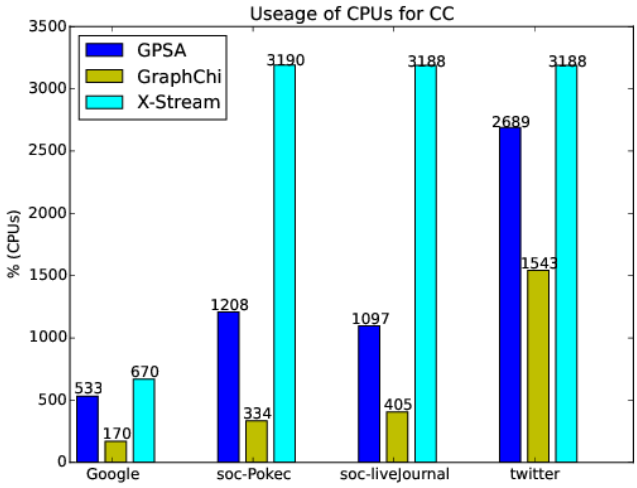



图 5.6 CC算法的多核CPU利用率

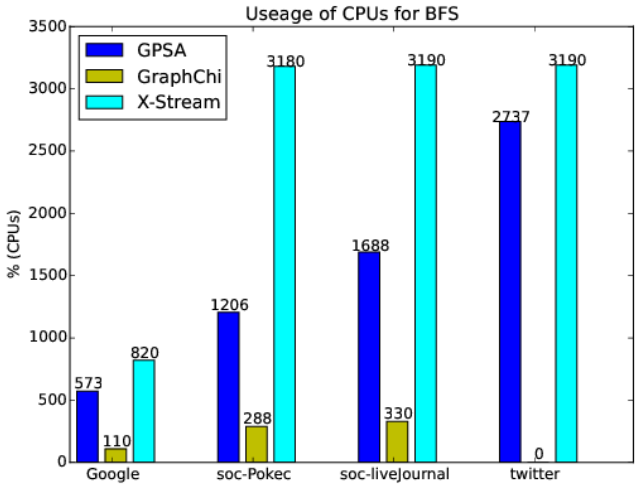



图 5.7 BFS算法的多核CPU利用率

数据量非常小，但是两者的CPU占用情况均高达99%。反观GraphChi与GPSA，它在处理大小不同的数据集时，对CPU的占用率则呈现出较好的扩展性，例如在图5.5中，GPSA在四个应用上对CPU的占用率分别为533%、1208%、1097%、2689%，当数据量逐步增大时，对CPU的占用率也会相应的增大。

首先，GraphChi的多核CPU的利用率相对较差的主要原因在于GraphChi需要写大量的数据到磁盘，涉及大量频繁的IO操作。虽然X-Stream同样需要写入大量的数据到磁盘，但是X-Stream利用异步IO来充分利用多核CPU的优势。其次，X-Stream具有较差的伸缩性。X-Stream是以边为中心的计算模型，内部实现使用线程并发，虽然利用异步IO提高了效率，但是也导致在X-Stream对CPU的占用率最高，缺乏一定的弹性。最后，GPSA在三个系统中不仅具有较高的CPU利用率，同时具有较好的伸缩性。这主要得益于GPSA的数据易变与不易变分离以及分发与计算分离，在节省大量IO操作的同时，Actor模型的轻量级并发使得在利用多核并发时相比线程并发而言更具优势。

## 5.4 小结

本章首先详细介绍了实验测试的软硬件环境与测试数据集，接着通过PageRank，连通分量、广度优先搜索三个应用在不同测试数据集上的运行情况与GraphChi和X-Stream进行性能测试与对比。通过效率和多核利用率两个方面的对比证明基于新模型的GPSA单机多核系统上的图处理系统不仅具有便捷高效的特点，同时还能充分发挥多核的优势。

# 结 论

本文在分析分布式图处理系统和单机图处理系统的基础上，对目前图处理系统中的一些缺陷进行讨论，并在传统BSP计算模型的基础上，利用Actor并发模型对BSP模型进行优化，提出了基于Actor并发模型的BSP计算模型。通过对Actor-BSP模型的数据行为进行分析和讨论，将图数据分为两个部分：常驻内存的顶点状态信息和保存在磁盘上的边数据信息，同时利用内存映射优化IO操作。实验证明，GPSA图处理系统不仅能够在单机多核系统上高效的处理大规模图，同时还能够充分发挥多核的优势。本文的主要工作包括以下几个方面：

1、本论文在做了大量调研的基础上，对目前大规模图处理系统进行分析和对比，详细讨论目前存在于分布式图处理系统的困难问题，说明在单机系统上进行大规模图处理的可行性，对单机系统上的图处理系统进行分析和对比，从计算模型的角度出发，对传统的BSP模型进行优化，提出崭新的Actor-BSP图计算模型。Actor-BSP模型中将传统以顶点为中心的模型中顺序执行的计算过程和分发过程分离解耦，同时使用Actor代替线程，并将计算过程和分发过程分布在不同的Actor上，提高任务并发量。同时，Actor-BSP简化了图计算的流程，由于淡化顶点作为整个计算的中心的概念，顶点之间消息的传递转换为Actor之间的消息分发，计算过程和分发过程之间通过消息建立联系，从而无需再计算中缓存大量的消息，节省大量的IO操作。另外，Actor-BSP模型缩短了单个任务的平均执行时间，提升计算效率。

2、在Actor-BSP模型的基础上对数据的访问行为进行分析和讨论，由于新模型消息发送的随机性，就无法避免计算Actor对顶点状态信息的随机访问，与其他单机系统尽力回避随机访问的做法不同，GPSA采用数据分离的方法，将图数据分为两个独立的部分：顶点的状态信息和图的结构信息。其中，顶点的状态信息按照顺序存储的方式利用内存映射技术将其映射到内存来支持随机访问，提升效率。而对于图的结构信息则保持于磁盘上，可以采用顺序访问的方式，进一步节省IO操作。

3. 独立完成了本文所论述的GPSA系统的调研，方案设计，具体编码实现和测试工作。列举PageRank、连通分量以及广度优先搜索三个常见图应用在GPSA系统上的实现，并将之与其他单机图处理系统从效率和多核利用率两个方面进行对比和分析，结果表明，GPSA不仅具有高效能够充分发挥多核优势的特点，而且具有较好的伸缩性与容错性。

虽然GPSA从改进计算模型角度出发，并取得良好的效果，但是GPSA依然存

在一些不足，有待进一步完善。

首先，GPSA不支持图结构改变的应用。GPSA将图分为两部分：顶点信息和边。其中，顶点常驻内存，支持随机访问和更新，但是边、边上的权重等信息保存在磁盘上，需要尽量避免随机读写。

其次，GPSA使用JAVA实现，在实现过程中为避免频繁的垃圾回收造成的性能影响，在消息的封装中使用基本类型，造成消息的生成仅仅支持数字类型，无法支持字符串或者对象等。

最后，GPSA的设计初衷是能够兼容以BSP为计算模型的大规模分布式图处理系统，希望能在将来能继续进行并开展在分布式方向的扩展和实验。

# 参考文献


[1] 李国杰, 程学旗. 大数据研究未来科技及经济社会发展的重大战略领域-大数据的研究现状与科学思考[J]. 中国科学院院刊, 2012, 27(6):647–657

[2] 严霄凤, 张德馨. 大数据研究. 计算机技术与发展, 2013, 4(32):4

[3] 陶雪娇, 胡晓峰, 刘洋. 大数据研究综述. Journal of System Simulmion, 2013.

[4] 李国杰. 大数据研究的科学价值. 中国计算机学会通讯, 2012, 8(9):8–15

[5] 孟小峰, 慈祥. 大数据管理: 概念, 技术与挑战. 计算机研究与发展, 2013, 50(1):146–169

[6] 王珊, 王会举, 覃雄派, et al. 架构大数据: 挑战, 现状与展望. 计算机学报, 2011, 34(10):1741–1752

[7] 张兴旺, 李晨晖, 秦晓珠. 云计算环境下大规模数据处理的研究与初步实现. 现代图书情报技术, 2011, 27(4):17–23

[8] 于戈, 谷峪, 鲍玉斌, et al. 云计算环境下的大规模图数据处理技术. 计算机学报, 2011, 34(10):1753–1767

[9] 余慧佳, 刘奕群, 张敏, et al. 基于大规模日志分析的搜索引擎用户行为分析. 中文信息学报, 2007, 21(1):109–114

[10] 朱珠. 基于Hadoop 的海量数据处理模型研究和应用[D]. 北京邮电大学, 2008, 37(5):47–49

[11] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, et al. Challenges in parallel graph processing. Parallel Processing Letters, 2007, 17(01):5–20

[12] Rishan Chen, Xuetian Weng, Bingsheng He, et al. Large graph processing in the cloud. In: Proc of Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, 1123–1126

[13] Ralf Lämmel. Google’s MapReduce programming model —Revisited. Science of computer programming, 2008, 70(1):1–30

[14] Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51(1):107–113

[15] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, et al. Pregel: a system for large-scale graph processing. In: Proc of Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, 135–146

[16] Semih Salihoglu, Jennifer Widom. Gps: A graph processing system. In: Proc of Proceedings of the 25th International Conference on Scientific and Statistical Database Management. ACM, 2013, 22

[17] U Kang, Hanghang Tong, Jimeng Sun, et al. Gbase: a scalable and general graph management system. In: Proc of Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011, 1091–1099

[18] U Kang, Hanghang Tong, Jimeng Sun, et al. Gbase: an efficient analysis platform for large graphs. The VLDB Journal, 2012, 21(5):637–650

[19] U Kang, Charalampos E Tsourakakis, Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In: Proc of Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on. IEEE, 2009, 229–238

[20] Leslie G Valiant. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8):103–111

[21] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, et al. Graphlab: A new framework for parallel machine learning. arXiv preprint. arXiv preprint arXiv:1006.4990, 2010, 1

[22] Bin Shao, Haixun Wang, Yatao Li. Trinity: A distributed graph engine on a memory cloud. In: Proc of Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013, 505–516

[23] Zuhair Khayyat, Karim Awara, Amani Alonazi, et al. Mizan: a system for dynamic load balancing in large-scale graph processing. In: Proc of Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013, 169–182

[24] Sungpack Hong, Hassan Chafi, Edic Sedlar, et al. Green-Marl: a DSL for easy and efficient graph analysis. In: Proc of ACM SIGARCH Computer Architecture News, volume 40. ACM, 2012, 349–362

[25] Sungpack Hong, Semih Salihoglu, Jennifer Widom, et al. Tech Report: Compiling Green-Marl into GPS, 2012

[26] Joseph E Gonzalez, Yucheng Low, Haijie Gu, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In: Proc of OSDI, volume 12. 2012, 2

[27] Raymond Cheng, Ji Hong, Aapo Kyrola, et al. Kineograph: taking the pulse of a fast-changing and connected world. In: Proc of Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012, 85–98

[28] Josep M Pujol, Vijay Erramilli, Georgos Siganos, et al. The little engine (s) that could: scaling online social networks. ACM SIGCOMM Computer Communication Review, 2011, 41(4):375–386

[29] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, et al. Managing Large Graphs on Multi-Cores with Graph Awareness.. In: Proc of USENIX Annual Technical Conference. 2012, 41–52

[30] Julian Shun, Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In: Proc of ACM SIGPLAN Notices, volume 48. ACM, 2013, 135–146

[31] Roger Pearce, Maya Gokhale, Nancy M Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In: Proc of Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010, 1–11

[32] Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In: Proc of Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013, 472–488

[33] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC.. In: Proc of OSDI, volume 12. 2012, 31–46

[34] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, et al. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: Proc of Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2013, 77–85

[35] 孙惠泉. 图论及其应用, volume 9. 科学出版社, 2004

[36] Sriram Srinivasan. A thread of one’s own. In: Proc of Workshop on New Horizons in Compilers, volume 4. 2006

[37] Rajesh K Karmani, Amin Shali, Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In: Proc of Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. ACM, 2009, 11–20

[38] Sriram Srinivasan, Alan Mycroft. Kilim: Isolation-typed actors for java. In: Proc of ECOOP 2008–Object-Oriented Programming. Springer, 2008: 104–128

[39] Sriram Srinivasan. Kilim: A Server Framework with Lightweight Actors Isolation Types Zero-copy Messaging: [Dissertation]. University of Cambridge, 2010

[40] 于慧彬, 齐鹏, 梁捷, et al. 内存映射文件在大数据量海洋调查数据处理中的应用. 海洋技术, 2010, 29(1):32–35

[41] 杨宁学, 诸昌钤, 聂爱丽, et al. 内存映射文件及其在大数据量文件快速存取中的应用[J]. 计算机应用研究, 2004, 21(8):187–188

[42] Richard M Karp. A survey of parallel algorithms for shared-memory machines. 1988.

# 致　谢

三年的研究生生活即将结束，我首先要感谢我的导师陈浩教授，本论文的完成得益于他的悉心指导和无倦教诲，论文从选题、构思、一直到修改都倾注了导师大量心血。陈老师严谨的治学态度、渊博的学识、博大的胸襟、一丝不苟的工作作风和平易近人的性格，都给我留下了深刻的印象。另外还要感谢孙建华副教授，她敏锐的洞察力以及对学术研究的极大热情令人敬佩。两位老师提倡的学术自由极大的激发了我们的研究热情和创新欲望，使我的科研水平以及综合素质得到提高。在论文完成之际，特向他们致以诚挚的敬意和衷心的感谢。

在论文的创作中还得到了许多同学的倾心帮助。常诚师兄在论文工作中给予了大量帮助、指导和有益讨论，促使我在学术上不断进步。肖军、谭元元同学作为实验室的小伙伴在业余时间相互探讨也给了我不少启迪，开阔了我的写作思路。师弟师妹们以及室友也给予了大量的帮助和支持，让我度过了一段难忘的时光，在此一并感谢，并祝你们成功！特别地，我要感谢我的家人，他们总是在背后默默的支持着我，一直记挂我的身体和学业，是他们给了我无穷的力量，让我迎接挑战，不断进步，祝福他们快乐，安康！

我还要感谢本论文所引文献的全部作者及所有为这篇论文做出贡献的人。

由于本人学术水平有限，论文中如有不妥之处，敬请各位老师批评、指正。最后，谨向在百忙之中抽出宝贵时间审阅本论文及答辩组的老师们致以最诚挚的谢意！

# 附录A　发表论文和参加科研情况说明

## （一）发表的学术论文

[1] XXX，XXX. Density and Non-Grid based Subspace Clustering via Kernel Density Estimation[C]. ECML-PKDD 2012, Bristol, UK.(Submitted, Under review)

[2] XXX，XXX. A tree parent storage based on hashtable for XML construction[C]. Communication Systems, Networks and Applications, Hongkong, 2010: 325-328. (EI DOI: 10.1109/ICCSNA.2010.5588732)

## （二）申请及已获得的专利（无专利时此项不必列出）

## （三）参与的科研项目

[1] 国家自然科学基金，基于程序分析方法的Web安全研究(61173166)，2012.1-2015.12

[2] 国家自然科学基金，GPU通用计算系统检查点方法研究(61272190)，2013.1-2016.12

[3] 新世纪优秀人才支持计划，基于多核不对称特性的虚拟机优化方法研究，2013.1-2015.12