获取教材和讲义 PPT 等各种课程资料请访问 http://dblab.xmu.edu.cn/node/422

=课程教材由林子雨老师根据网络资料编著=

**厦门大学计算机科学系教师 林子雨 编著**

http://www.cs.xmu.edu.cn/linziyu

2013 年 9 月

# 前言

本教程由厦门大学计算机科学系教师林子雨编著，可以作为计算机专业研究生课程《大数据技术基础》的辅助教材。

本教程的主要内容包括：大数据概述、大数据处理模型、大数据关键技术、大数据时代面临的新挑战、NoSQL 数据库、云数据库、Google Spanner、Hadoop、HDFS、HBase、MapReduce、Zookeeper、流计算、图计算和 Google Dremel 等。

本教程是林子雨通过大量阅读、收集、整理各种资料后精心制作的学习材料，与广大数据库爱好者共享。教程中的内容大部分来自网络资料和书籍，一部分是自己撰写。对于自写内容，林子雨老师拥有著作权。

本教程 PDF 文档及其全套教学 PPT 可以通过网络免费下载和使用（下载地址：http://dblab.xmu.edu.cn/node/422）。教程中可能存在一些问题，欢迎读者提出宝贵意见和建议！

本教程已经应用于厦门大学计算机科学系研究生课程《大数据技术基础》，欢迎访问 2013 班级网站 http://dblab.xmu.edu.cn/node/423。

林子雨的 E-mail 是：ziyulin@xmu.edu.cn。

林子雨的个人主页是：http://www.cs.xmu.edu.cn/linziyu。

林子雨于厦门大学海韵园

2013 年 9 月

# 第 9 章 图计算

**厦门大学计算机科学系教师 林子雨 编著**

个人主页：http://www.cs.xmu.edu.cn/linziyu

课程网址：http://dblab.xmu.edu.cn/node/422

2013 年 9 月

# 第 9 章 图计算

随着大数据时代的到来，图的规模越来越大，有的甚至有数十亿的顶点和数千亿的边，这就给高速地处理图数据带来了挑战。一台机器已经不能存放所有需要计算的数据了，所以需要一个分布式的计算环境。而已有的图计算框架和图算法库不能很好地满足计算需求，因此，新的图计算框架应运而生。

本章内容首先简单介绍了图计算，然后详细介绍了当前热门的 Google 图计算框架 Pregel，包括 Pregel 图计算模型、Pregel 中的 C++ API、Pregel 的执行过程和 Pregel 的算法实现，内容要点如下：

- 图计算简介
- Google Pregel 简介
- Google Pregel 图计算模型
- Pregel 的 C++ API
- Pregel 模型的基本体系结构
- Pregel 模型的应用实例
- 改进的图计算模型

## 9.1 图计算简介

在实际应用中，存在许多图计算问题，比如最短路径、集群、网页排名、最小切割、连通分支等等。图计算算法的性能，直接关系到应用问题解决的高效性，尤其对于大型图（比如社交网络和网络图）而言，更是如此。下面我们首先指出传统图计算解决方案的不足之处，然后介绍两大类通用图计算软件。

### 9.1.1 传统图计算解决方案的不足之处

在很长一段时间内，都缺少一个可扩展的通用系统来解决大型图的计算问题。很多传统的图计算算法都存在以下几个典型问题：（1）常常表现出比较差的内存访问局部性；（2）针

对单个顶点的处理工作过少；（3）计算过程中伴随着并行度的改变。

针对大型图（比如社交网络和网络图）的计算问题，可能的解决方案及其不足之处具体如下：

- 为特定的图应用定制相应的分布式实现。不足之处是，在面对新的图算法或者图表示方式时，就需要做大量的重复实现，不通用。
- 基于现有的分布式计算平台进行图计算。但是，在这种情况下，它们往往并不适于做图处理。比如，MapReduce 就是一个对许多大规模计算问题都非常合适的计算框架。有时，它也被用来对大规模图对象进行挖掘，但是，通常在性能和易用性上都不是最优的。尽管这种对数据处理的基本模式经过扩展，已经可以使用方便的聚合以及类似于 SQL 的查询方式，但是，这些扩展对于图算法这种更适合用消息传递模型的问题来说，通常并不理想。
- 使用单机的图算法库。比如 BGL、LEAD、NetworkX、JDSL、Standford GraphBase 和 FGL 等等；但是，这种方式对可以解决的问题的规模提出了很大的限制。
- 使用已有的并行图计算系统。Parallel BGL 和 CGMgraph 这些库实现了很多并行图算法，但是，并没有解决对大规模分布式系统中来说非常重要的容错等一些问题。

## 9.1.2 图计算通用软件

正是因为传统的图计算解决方案无法解决大型图的计算问题，因此，就需要设计能够用来解决这些问题的通用图计算软件。针对大型图的计算，目前通用的图处理软件主要包括两种：第一种主要是基于遍历算法和实时的图数据库，如 Neo4j 、OrientDB、DEX 和 InfiniteGraph。第二种则是以图顶点为中心的消息传递批处理的并行引擎，如 Hama、Golden Orb、Giraph 和 Pregel。

**第一种图处理软件**，基本都基于 Tinkerpop 的图基础框架，Tinkerpop 项目关系如图 9-1 所示。

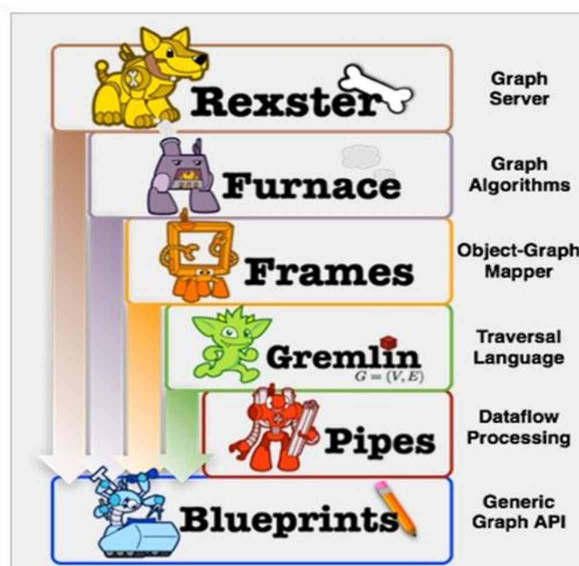


图 9-1 Tinkerpop 项目关系图

下面具体介绍一下 TinkerPop 框架的各层功能：

- **Blueprints**：是一组针对属性图数据模型的接口、实现、测试套件，它和 JDBC 类似，但是，它是基于图数据库的。就其本身而言，它提供了一组通用的接口，允许开发者对其图数据库后台即插即用。另外，在 Blueprints 上编写的软件可以运行于所有的 Blueprints 开启的图数据库。在 Tinkerpop 的图基础框架中，Blueprints 为其他几层提供基础技术服务。
- **Pipes**：是一个应用流程图的数据流框架。一个流程图由很多通过“通信边”相连的 pipe 顶点组成。一个 pipe 实现了一个简单的计算步骤，它可以和其他的 pipe 相组合，一起产生一个更大的计算。这样的数据流图允许拆分、合并、循环和输入输出数据的相互转换。伴随着主 Pipe 的分布，会产生大量的 Pipe 类。只要了解了每个 Pipe 的实现，就可以直接使用 Pipe 框架。
- **Gremlin**：是一种图遍历语言，可以用于图的查询、分析和处理。它工作在那些执行 Blueprints 性能图数据模型的图数据库和框架上。Gremlin 是能用于各种 JVM 语言的一种图遍历。Gremlin 的布局为 Java 和 Groovy 提供了支持。
- **Frames**：把 Blueprints 图映射为一组相互关联的域对象集。Frames 中经常使用 InvocationHandler、Porxy 类和 Annotations，使开发者可以用一个特殊的 Java 接口来构造一个图元素（顶点或边）。通过 Frames，非常容易确认图中数据各自的图解，对应哪一个带有注释的 Java 接口。
- **Furnace**：是能启用 Blueprints 图的一个算法包。在图理论和分析的历史进程中开发

了很多的图算法。这些算法中的大多数是为无标号的、单一关系的图而设计的。Furnace 的目的就是在单一关系图算法中揭示属性图（像属性化图或多关系图）。另外，Furnace 提供了针对不同图计算场景，各种图算法的优化实现，像单机图和分布图。

- **Rexster**：是一个图服务器，通过 REST 和一个被称为 RexPro 的二进制协议展现任意的 Blueprints 图。HTTP 网页服务器提供了标准的低层 GET、POST、PUT 和 DELETE 方法，一个灵活的、像开发一个外部服务器（如通过 Gremlin 的特殊图查询）一样允许插件法的扩展模型，用 Gremlin 编写的服务器端存储程序和一个基于浏览器的接口——Dog House。Rexster Console 使对 Rexster 服务器中的配置图的远程脚本评估成为可能。Rexster Kibbles 是由 TinkerPop 提供的各种 Rexster 服务器的扩展集。

**第二种图处理软件**，则主要是基于 BSP 模型所实现的并行图处理包。BSP 是由哈佛大学 Viliant 和牛津大学 Bill McColl 提出的并行计算模型，全称为“整体同步并行计算模型”(Bulk Synchronous Parallel Computing Model,简称 BSP 模型),又名“大同步模型”。创始人希望 BSP 模型像冯 诺伊曼体系结构那样，架起计算机程序语言和体系结构间的桥梁，故又称作“桥模型” (Bridge Model)。一个 BSP 模型由大量相互关联的处理器所组成，它们之间形成了一个通信网络。每个处理器都有快速的本地内存和不同的计算线程。一次 BSP 计算过程包括一系列全局超步，所谓的超步就是计算中的一次迭代。每个超步主要包括三个组件：

- **并发计算**：每个参与的处理器都有自身的计算任务，它们只读取存储在本地内存的值，这些计算都是异步并且独立的；
- **通讯**：处理器群相互交换数据，交换的形式是，由一方发起推送(put)和获取(get)操作；
- **栅栏同步**(Barrier synchronisation): 当一个处理器遇到“路障”，会等到其他所有处理器完成它们的计算步骤；每一次同步也是一个超步的完成和下一个超步的开始；图 9-2 是一个超步的垂直结构图。

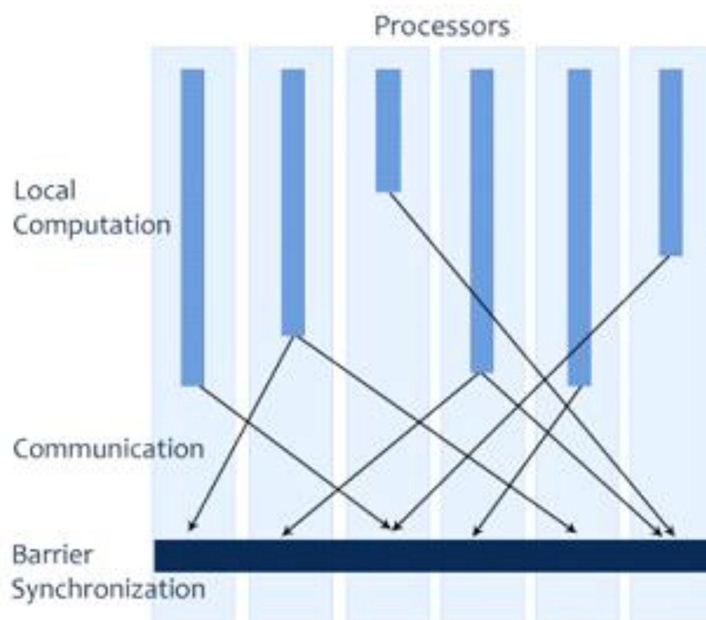


图 9-2 一个超步的垂直结构图

## 9. 2 Google Pregel 简介

Pregel就是一种基于BSP模型所实现的并行图处理包。为了解决大型图的分布式计算问题，Pregel搭建了一套可扩展的、有容错机制的平台，该平台提供了一套非常灵活的API，可以描述各种各样的图计算。Pregel是一个用于分布式图计算的计算框架，主要用于图遍历（BFS）、最短路径（SSSP）、PageRank计算等等。共享内存的运行库有很多，但是，对于Google来说，一台机器早已经放不下需要计算的数据了，所以，需要分布式这样一个计算环境。

没有Pregel之前，你可以选择用MapReduce来做，但是效率很低。图算法如果用MapReduce实现，需要一系列的MapReduce的调用。从一个阶段到下一个阶段，它需要传递整个图的状态，会产生大量不必要的序列化和反序列化开销。而Pregel使用超步简化了这个过程。下面我们以一个实例来阐述采用Pregel和MapReduce来执行图计算的区别。

**实例：PageRank算法在Pregel和MapReduce中的实现。**

PageRank 算法作为 Google 的网页链接排名算法，具体公式如下：

$$PR = \beta\sum_{i=1}^{n}\frac{PR_i}{N_i} + (1-\beta)\frac{1}{N}$$

对于任意一个链接，其PR值为链入到该链接的源链接的PR值对该链接的贡献和（分母

Ni为第i个源链接的链出度)。

Pregel是Google提出的专门为图计算所设计的计算模型，主要来源于BSP并行计算模型的启发。要用Pregel计算模型实现PageRank算法，也就是将网页排名算法映射到图计算中，这其实是很自然的，因为，网络链接是一个连通图。

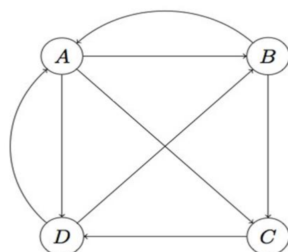


图 9-3 一个连通图

图9-3就是四个网页（A,B,C,D）互相链入链出组成的连通图。根据Pregel的计算模型，将计算定义到顶点（即A，B，C，D）上来，每个顶点对应一个对象，即一个计算单元。

每一个计算单元包含三个成员变量：

- Vertex value：顶点对应的PR值；
- Out edge：只需要表示一条边，可以不取值；
- Message：传递的消息，因为需要将本顶点对其它顶点的PR贡献值，传递给目标顶点。

每一个计算单元包含一个成员函数：

- Compute：该函数定义了顶点上的运算，包括该顶点的PR值计算，以及从该顶点发送消息到其链出顶点。

PageRank算法的Pregel实现代码如下：

```
class PageRankVertex
        : public Vertex<double, void, double> {
public:
        virtual void Compute(MessageIterator* msgs) {
                if (superstep() >= 1) {
                        double sum = 0;
                        for (; !msgs->Done(); msgs->Next())
                        sum += msgs->Value();
                        *MutableValue() =
                                0.15 / NumVertices() + 0.85 * sum;
                }
                if (superstep() < 30) {
                        const int64 n = GetOutEdgeIterator().size();
```

```
                SendMessageToAllNeighbors(GetValue() / n);
            } else {
                VoteToHalt();
            }
        }
};
```

PageRankVertex 继承自 Vertex 类。顶点 value 类型是 double，用来保存 PageRank 中间值，消息类型也是 double，用来传输 PageRank 分数，边的 value 类型是 void，因为不需要存储任何信息。我们假设，在第 0 个超步时，图中各顶点的 value 值被初始化为 1/NumVertices()。在前 30 个超步中，每个顶点都会沿着它的出射边，发送它的 PageRank 值除以出射边数目以后的结果值。从第 1 个超步开始，每个顶点会将到达的消息中的值加到 sum 值中，同时将它的 PageRank 值设为 0.15/ NumVertices()+0.85*sum。到了第 30 个超步后，就没有需要发送的消息了，同时所有的顶点 VoteToHalt。在实际中，PageRank 算法需要一直运行直到收敛，可以使用 aggregators 来检查是否满足收敛条件。

MapReduce 也是 Google 提出的一种计算模型，它是为全量计算而设计。采用 MapReduce 实现 PageRank 的计算过程需要包括以下三个阶段：

- **阶段 1：解析网页**

Map task 把（URL，page content）映射为（URL，（PRinit，list-of-urls）），其中，PRinit 是 URL 的“seed”PageRank，list-of-urls 包含了该 URL 页面中的外链所指向的所有页。Reduce task 只是恒等函数。

在阶段 1 中，对于 MapReduce 程序，Map 函数的输入是用户指定的文件（这里就是一个网页），那么，输入的 value 值就是网页内容，key 值是网页的 url。程序对输入的 key 和 value 进行一系列操作，然后，按(key,value)键值对的形式输出。系统获取 Map 函数的输出，把相同的 key 合并，再把 key 和 value 集合作为键值对作为 reduce 函数的输入。程序员自定义 reduce 函数中的处理方法（这里是一个恒等函数），输出(key,value)键值对到磁盘文件（这里的键值对中的 key 仍是该网页的 url，value 是该页的初始 PR 值和链出页的 url 列表）。

- **阶段 2：PageRank 分配**

Map task 得到（URL，（cur_rank，url_list）），对于每一个 url_list 中的 u，输出（u，cur_rank/|url_list|），并输出链接关系（URL，url_list），用于迭代。

Reduce task 获得（URL，url_list）和很多（URL，var）值对，对于具有相同 key 值的

value 进行汇总，并把汇总结果乘以 d(这里是 0.85)，然后输出输出（URL，（new_rank，url_list））。

阶段 2 是一个迭代过程，每次将磁盘上的文件读入，key 为每一个网页链接，value 的第一项是当前网页的 PR 值，第二项是该网页中的外链列表。Map 函数将输入的每一对(key,value)“反转”成多对(value,key)输出，也就是说，每个输出的 key 为输入 value 中的每一个外链，而输出的 value 则为输入 key 的链接、输入 key 的 PR 值除以输入 key 的对外链接数。

在 Reduce 函数中，就可以根据输入的 value 中的参数来计算每一个 key 新的 PageRank 值，并按 Map 函数的格式输出到磁盘，以作为下一次迭代的输入。迭代多次后，PageRank 值趋于稳定，就得出了较为精确的 PageRank 值。

- **阶段 3：最后阶段**

一个非并行组件决定是否达到收敛。如果达到收敛，写出 PageRank 生成的列表。否则，回退到第 2 阶段的输出，进行另一个第 2 阶段的迭代。

下面具体解释一下阶段 2 的伪代码实现：

**Mapper 函数的伪码：**

```
input <PageN, RankN> -> PageA, PageB, PageC ... // 链接关系
begin
    Nn := the number of outlinks for PageN;
    for each outlink PageK
        output PageK -> <PageN, RankN/Nn>
    // 同时输出链接关系，用于迭代
    output PageN -> PageA, PageB, PageC ...
end
```

Mapper 的输出如下（已经排序，所以 PageK 的数据排在一起，最后一列则是链接关系对）：

```
PageK -> <PageN1, RankN1/Nn1>
PageK -> <PageN2, RankN2/Nn2>
...
PageK -> <PageAk, PageBk, PageCk>
```

**Reduce 函数的伪码：**

```
input mapper's output
begin
    RankK := 0;
    for each inlink PageNi
        RankK += RankNi/Nni * beta
    // output the PageK and its new Rank for the next iteration
    output <PageK, RankK> -> <PageAk, PageBk, PageCk...>
end
```

上述伪代码只是一次迭代的代码，多次迭代需要重复运行，需要说明的是这里可以优化的地方很多，比如把 Mapper 的<pageN,pageA-pageB-pageC…>内容缓存起来，这样就不用再 output 作为 Reducer 的 input 了，同时，Reducer 在 output 的时候也不用传递同样的<pagek,pageO-pageP-pageQ>，减少了大量的 I/O，因为，在 PageRank 计算是，类似这样的数据才是主要数据。

简单地来讲，Pregel 将 PageRank 处理对象看成是连通图，而 MapReduce 则将其看成是 Key-Value 对。Pregel 将计算细化到顶点 vertex，同时在 vertex 内控制循环迭代次数，而 MapReduce 则将计算批量化处理，按任务进行循环迭代控制。图算法如果用 MapReduce 实现，需要一系列的 MapReduce 的调用。从一个阶段到下一个阶段，它需要传递整个图的状态，会产生大量不必要的序列化和反序列化开销。而 Pregel 使用超步简化了这个过程。

# 9.3 Google Pregel 图计算模型

## 9.3.1 Pregel 的消息传递模型

Pregel 选择了一种纯消息传递模型（如图 9-4 所示），忽略远程数据读取和其他共享内存的方式。这样做的原因有两个：第一，消息传递有足够的表达能力，没必要使用远程读取，还没有发现哪种算法是消息传递所不能表达的；第二是出于性能的考虑，在一个集群环境中，从远程机器上读取一个值是会有很高的延迟的，这种情况很难避免，而 Pregel 的消息传递模式通过异步和批量的方式传递消息，可以缓解这种远程读取的延迟。

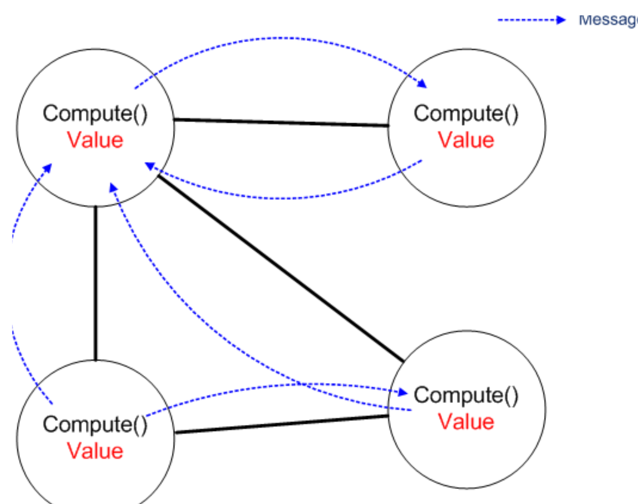


图 9-2 纯消息传递模型图

## 9.3.2 Pregel 的计算过程

Pregel 的计算过程是由一系列被称为超步(superstep)的迭代(iterations)组成的。在每一个超步中，计算框架都会针对每个顶点调用用户自定义的函数，这个过程是并行的。该函数描述的是一个顶点 V 在一个超步 S 中需要执行的操作。该函数可以读取前一个超步(S-1)中发送给 V 的消息，并发送消息给其他顶点，这些消息将会在下一个超步(S+1)中被接收，并且在此过程中修改顶点 V 及其出射边的状态。消息通常沿着顶点的出射边发送，但一个消息可能会被发送到任意已知 ID 的顶点上去。

## 9.3.3 Pregel 计算模型的实体

在 Pregel 计算模型中，输入是一个有向图，该有向图的每一个顶点都有一个相应的由 String 描述的顶点标识符。每一个顶点都有一个与之对应的可修改的用户自定义值。每一条有向边都和其源顶点关联，并且也拥有一个可修改的用户自定义值，并同时记录了其目标顶点的标识符。

在每个超步中，顶点的计算都是并行的，每个顶点执行相同的用于表达给定算法逻辑的用户自定义函数。每个顶点可以修改其自身及其出射边的状态，接收前一个超步(S-1)中发送给它的消息，并发送消息给其他顶点(这些消息将会在下一个超步中被接收)，甚至是修改整个图的拓扑结构。在这种计算模式中，“边”并不是核心对象，没有相应的计算运行在其上。

## 9.3.4 Pregel 计算模型的进程

算法是否能够结束，取决于是否所有的顶点都已经“vote”标识其自身已经达到“halt”状态了。在第 0 个超步，所有顶点都处于 active 状态，所有的 active 顶点都会参与所有对应超步中的计算。顶点通过将其自身的 status 设置成“halt”来表示它已经不再 active。这就表示该顶点没有进一步的计算需要去执行，除非再次被外部触发，而 Pregel 框架将不会在接下来的超步中执行该顶点，除非该顶点收到其它顶点传送的消息。如果顶点接收到消息被唤醒进入 active 状态，那么在随后的计算中该顶点必须显式地“deactive”。整个计算在所有顶点都达到“inactive”状态，并且没有 message 在传送的时候才宣告结束。这种简单的状态机如图 9-5 所示。

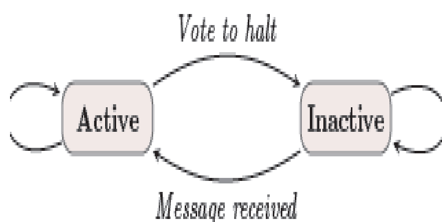


图 9-5 一个简单的状态机图

下面通过一个简单的例子来说明这些基本概念。

给定一个强连通图（如图 9-6 所示），图中每个顶点都包含一个值，它会将最大值传播到每个顶点。在每个超步中，顶点会从接收到的消息中选出一个最大值，并将这个值传送给其所有的相邻顶点。当某个超步中已经没有顶点更新其值，那么算法就宣告结束。

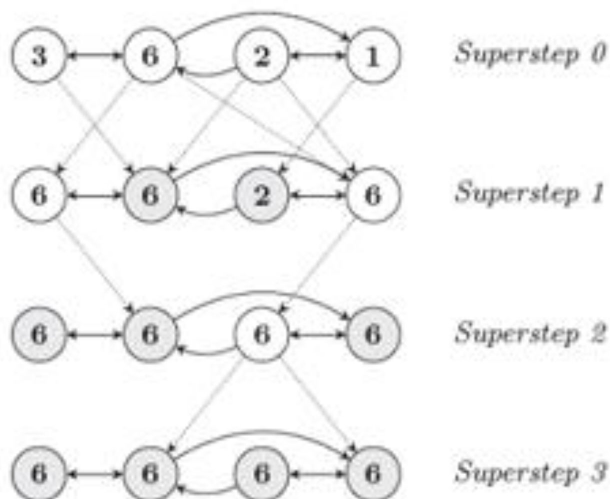


图 9-6 一个求最大值的步骤图

# 9.4 Pregel 的 C++ API

编写一个 Pregel 程序需要继承 Pregel 中已预定义好的一个基类——Vertex 类，如下所示：

```
template <typename VertexValue, typename EdgeValue, typename MessageValue>
class Vertex {
  public:
    virtual void Compute(MessageIterator* msgs) = 0;
    const string& vertex_id() const;
    int64 superstep() const;
    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                const MessageValue& message);
    void VoteToHalt();
};
```

该类的模板参数中定义了三个值类型参数，分别表示顶点、边和消息。每一个顶点都有一个对应的给定类型的值。这种形式可能看上有很多限制，但用户可以用 protocol buffer 来管理增加的其他定义和属性。而边和消息类型的行为比较类似。

用户覆写 Vertex 类的虚函数 Compute()，该函数会在每一个超步中对每一个顶点进行调用。预定义的 Vertex 类方法允许 Compute()方法查询当前顶点及其边的信息，以及发送消息到其他的顶点。Compute()方法可以通过调用 GetValue()方法来得到当前顶点的值，或者通过调用 MutableValue()方法来修改当前顶点的值。同时还可以通过由出射边的迭代器提供的方法来查看修改出射边对应的值。这种状态的修改是立时可见的。由于这种可见性仅限于被修改的那个顶点，所以，不同顶点并发进行的数据访问是不存在竞争关系的。

顶点和其对应的边所关联的值，是唯一需要在超步之间持久化的顶点级状态。将由计算框架管理的图状态限制在一个单一的顶点值或边值的这种做法，简化了主计算流程、图的分布以及故障恢复。

## 9.4.1消息传递机制

顶点之间的通信是直接通过发送消息来实现的，每条消息都包含了消息值和目标顶点的名称。消息值的数据类型是由用户通过 Vertex 类的模版参数来指定。

在一个超步中，一个顶点可以发送任意多的消息。当顶点 V 的 Compute()方法在 S+1 超步中被调用时，所有在 S 超步中发送给顶点 V 的消息都可以通过一个迭代器来访问到。在该迭代器中并不保证消息的顺序，但是，可以保证消息一定会被传送并且不会重复。

一种通用的使用方式为：对一个顶点 V，遍历其自身的出射边，向每条出射边发送消息到该边的目标顶点，如 PageRank 算法所示的那样。但是，消息要发送到的目标顶点 dest_vertex，并不一定是顶点 V 的相邻顶点。一个顶点可以从之前收到的消息中获取到其非相邻顶点的标识符，或者顶点标识符可以隐式地得到。比如，图可能是一个 clique(一个图中两两相邻的一个点集，或是一个完全子图)，顶点的命名规则都是已知的(从 V1 到 Vn)，在这种情况下甚至都不需要显式地保存边的信息。

当任意一个消息的目标顶点不存在时，便执行用户自定义的 handlers。比如在这种情况下，一个 handler 可以创建该不存在的顶点或从源顶点中删除这条边。

找最大值的代码实现如下：

```
Class MaxFindVertex
        : public Vertex<double, void, double> {
    public:
        virtual void Compute(MessageIterator* msgs) {
            int currMax = GetValue();
            SendMessageToAllNeighbors(currMax);
            for ( ; !msgs->Done(); msgs->Next()) {
                if (msgs->Value() > currMax)
                    currMax = msgs->Value();
            }
            if (currMax > GetValue())
                *MutableValue() = currMax;
            else VoteToHalt();
```

```
                    }
};
```

## 9.4.2 Combiner

发送消息，尤其是当目标顶点在另外一台机器上时，会产生一些开销。某些情况可以在用户的协助下降低这种开销。比方说，假如 Compute() 收到许多的 int 值消息，而它仅仅关心的是这些值的和，而不是每一个 int 的值，这种情况下，系统可以将发往同一个顶点的多个消息合并成一个消息，该消息中仅包含它们的“和值”，这样就可以减少传输和缓存的开销。

Combiners 在默认情况下并没有被开启，这是因为要找到一种对所有顶点的 Compute() 函数都合适的 Combiner 是不可能的。而用户如果想要开启 Combiner 的功能，需要继承 Combiner 类，覆写其 virtual 函数 Combine()。框架并不会确保哪些消息会被 Combine 而哪些不会，也不会确保传送给 Combine()的值和 Combining 操作的执行顺序。所以，Combiner 只应该对那些满足交换律和结合律的操作才给予打开。

对于某些算法来说，比如单源最短路径，我们观察到通过使用 Combiner 可以把流量降低 4 倍多。

下面是有关 combiner 应用的例子。假设我们想统计在一组相关联的页面中所有页面的链接数。在第一个迭代中，对从每一个顶点（页面）出发的链接，我们会向目标页面发送一个消息。这里输入消息上的 count 函数可以通过一个 combiner 来优化性能。在上面求最大值的例子中，一个 Max combiner 可以减少通信负荷（如图 9-7 所示），比如，假设顶点 1 和 6 在一台机器上，顶点 2 和 3 在另一台机器上，顶点 3 向顶点 6 传递的值是 3，顶点 2 向顶点 6 传递的值是 2，顶点 2 和顶点 3 都需要把消息传递到目标顶点 6，因此，可以采用 Combine() 函数把这两个消息进行合并后再发送给目标顶点 6，这样就减少了网络通信负责。

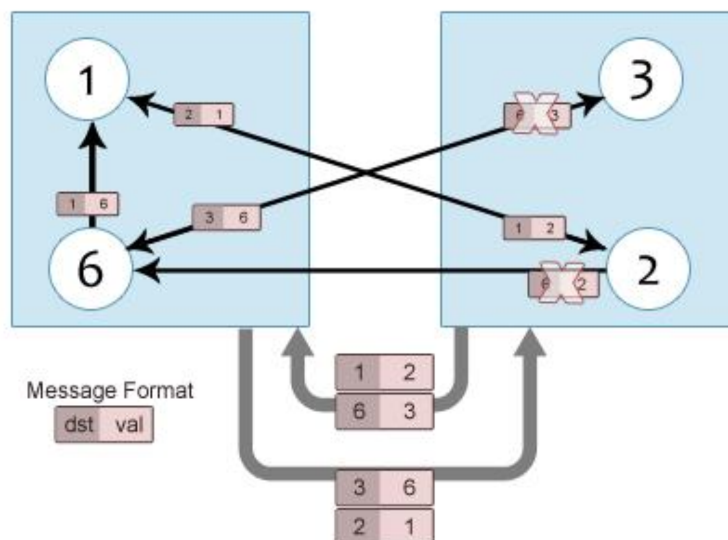


图 9-7 combiner 应用的例子

## 9. 4. 3 Aggregator

Pregel 的 Aggregators 是一种提供全局通信、监控和数据查看的机制。在一个超步 S 中，每一个顶点都可以向一个 Aggregator 提供一个数据，系统会使用一种 reduce 操作来负责聚合这些值，而产生的值将会对所有的顶点在超步 S+1 中可见。Pregel 包含了一些预定义的 aggregators，比如可以在各种整数和 string 类型上执行的 min、max、sum 操作。

Aggregators 可以用来做统计。例如，一个 sum aggregator 可以用来统计每个顶点的出度，最后相加就是整个图的边的条数。更复杂的一些 reduce 操作还可以产生统计直方图。

Aggregators 也可以用来做全局协同。例如，Compute()函数的一些逻辑分支可能在某些超步中执行，直到当“and” aggregator 表明所有顶点都满足了某条件时，会去执行另外的逻辑分支直到结束。又比如一个作用在顶点 ID 之上的 min 和 max aggregator，可以用来选定某顶点在整个计算过程中扮演某种角色等。

要定义一个新的 aggregator，用户需要继承预定义的 Aggregator 类，并定义在第一次接收到输入值后如何初始化，以及如何将接收到的多个值最后 reduce 成一个值。Aggregator 操作也应该满足交换律和结合律。

默认情况下，一个 aggregator 仅仅会对来自同一个超步的输入进行聚合，但是，有时也可能需要定义一个 sticky aggregator，它可以从所有的超步中接收数据。这是非常有用的，比如要维护全局的边条数，那么就仅仅在增加和删除边的时候才调整这个值了。

还可以有更高级的用法。比如，可以用来实现一个△-stepping 最短路径算法所需要的分

布式优先队列。每个顶点会根据它的当前距离分配一个优先级 bucket。在每个超步中，顶点将它们的 indices 汇报给 min aggregator。在下一个超步中，将最小值广播给所有 worker，然后让在最小 index 的 bucket 中的顶点放松它们的边。

一个有关 aggregator 应用的例子：Sum 运算符应用于每个顶点的出射边数，可以用来生成图中边的总数并使它能与所有的顶点相通信。

更复杂的规约运算符甚至可以产生直方图。在求最大值得例子中，我们可以通过运用一个 Max aggregator 在一个超步中完成整个程序。

### 9.4.4 Topology mutation

有一些图算法可能需要改变图的整个拓扑结构。比如一个聚类算法，可能会将每个聚类替换成一个单一顶点，又比如一个最小生成树算法，会删除所有除了组成树的边之外的其他边。正如用户可以在自定义的 Compute()函数能发送消息，同样可以产生在图中增添和删除边或顶点的请求。

多个顶点有可能会在同一个超步中产生冲突的请求(比如两个请求都要增加一个顶点 V，但初始值不一样)。Pregel 中用两种机制来决定如何调用：局部有序和 handlers。

由于是通过消息发送的，拓扑改变在请求发出以后，在超步中可以高效地执行。在该超步中，删除会首先被执行，先删除边，后删除顶点，因为顶点的删除通常也意味着删除其所有的出射边。然后执行添加操作，先增加顶点，后增加边，并且所有的拓扑改变都会在 Compute()函数调用前完成。这种局部有序保证了大多数冲突的结果的确定性。

剩余的冲突就需要通过用户自定义的 handlers 来解决。如果在一个超步中有多个请求需要创建一个相同的顶点，在默认情况下系统会随便挑选一个请求，但有特殊需求的用户可以定义一个更好的冲突解决策略，用户可以在 Vertex 类中通过定义一个适当的 handler 函数来解决冲突。同一种 handler 机制将被用于解决由于多个顶点删除请求或多个边增加请求或删除请求而造成的冲突。Pregel 委托 handler 来解决这种类型的冲突，从而使得 Compute()函数变得简单，而这样同时也会限制 handler 和 Compute()的交互，但这在应用中还没有遇到什么问题。

Pregel 的协同机制比较懒，全局的拓扑改变在被 apply 之前不需要进行协调，即在变更请求的发出端不会进行任何的控制协调，只有在它被接收到然后 apply 时才进行控制，这样就简化了流程，同时能让发送更快。这种设计的选择是为了优化流式处理。直观来讲，就是

对顶点 V 的修改引发的冲突由 V 自己来处理。

Pregel 同样也支持纯 local 的拓扑改变，例如，一个顶点添加或删除其自身的出射边或删除其自己。Local 的拓扑改变不会引发冲突，并且顶点或边的本地增减能够立即生效，很大程度上简化了分布式编程。

### 9.4.5 Input and Output

Pregel 可以采用多种文件格式进行图的保存，比如可以用 text 文件、关系数据库或者 Bigtable 中的行。为了避免规定死一种特定文件格式，Pregel 将“从输入中解析出图结构”这个任务从图的计算过程中进行了分离。类似地，结果可以以任何一种格式输出，并根据应用程序选择最适合的存储方式。Pregel library 本身提供了很多常用文件格式的 readers 和 writers，但是，用户可以通过继承 Reader 和 Writer 类来定义他们自己的读写方式。

## 9.5 Pregel 的基本体系结构

Pregel 是为 Google 的集群架构而设计的。每一个集群都包含了上千台机器，这些机器都分列在许多机架上，机架之间有着非常高的内部通信带宽。集群之间是内部互联的，但地理上是分布在不同地方的。

应用程序通常通过一个集群管理系统来执行，该管理系统会通过调度作业来优化集群资源的使用率，有时候会杀掉一些任务或将任务迁移到其他机器上去。该系统中提供了一个名字服务系统，所以，各任务间可以通过与物理地址无关的逻辑名称来各自标识自己。持久化的数据被存储在 GFS 或 Bigtable 中，而临时文件比如缓存的消息则存储在本地磁盘中。

### 9.5.1 Pregel 的执行过程

Pregel library 将一张图划分成许多的 partitions（如图 9-8 所示），每一个 partition 包含了一些顶点和以这些顶点为起点的边。将一个顶点分配到某个 partition 上去，取决于该顶点的 ID，这意味着，即使在别的机器上，也是可以通过顶点的 ID 来知道该顶点是属于哪个 partition，即使该顶点已经不存在了。默认的 partition 函数为 hash(ID) mod N，N 为所有 partition 总数，但是，用户可以替换掉它。

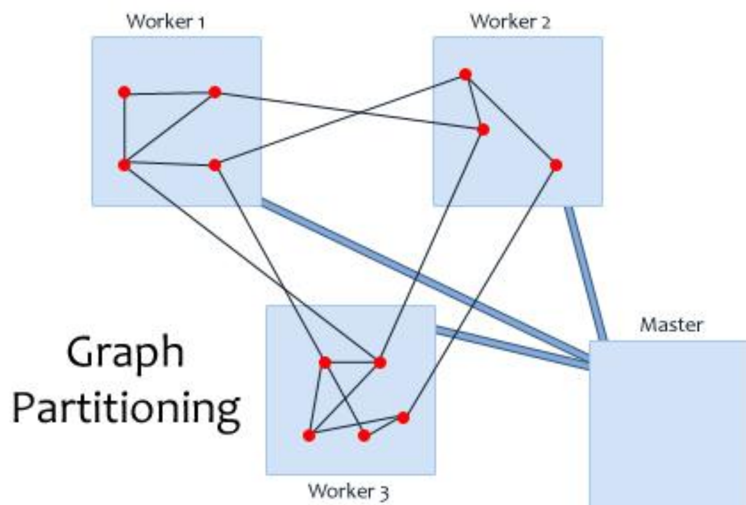


图 9-8 图的划分图

将一个顶点分配给哪个 worker 机器，是整个 Pregel 中对分布式不透明的主要地方。有些应用程序使用默认的分配策略就可以工作得很好，但是，有些应用可以通过定义一些可以更好地利用图本身的 locality 的分配函数而从中获益。比如，一种典型的可以用于 Web graph 的启发式方法是，将来自同一个站点的网页数据分配到同一台机器上进行计算。

在不考虑出错的情况下，一个 Pregel 程序的执行过程（如图 9-9 和图 9-10 所示）分为如下几个步骤：

1. 用户程序的多个 copy 开始在集群中的机器上执行。其中，有一个 copy 将会作为 master，其他的作为 worker，master 不会被分配图的任何一部分，而只是负责协调 worker 间的工作。worker 利用集群管理系统中提供的名字服务，来定位 master 位置，并发送注册信息给 master。

2. Master 决定对这个图需要多少个 partition，并分配一个或多个 partitions 到 worker 所在的机器上，如图 9-8 所示。这个数字也可能由用户进行控制。一个 worker 上有多个 partition 的情况下，可以提高 partitions 间的并行度，实现更好的负载平衡，通常都可以提高性能。每一个 worker 负责维护在其之上的图的那一部分的状态(顶点及边的增删)，对该部分中的顶点执行 Compute()函数，并管理发送出去的以及接收到的消息。每一个 worker 都知道该图的计算在所有 worker 中的分配情况。

3. Master 进程为每个 worker 分配用户输入中的一部分，这些输入被看作是一系列记录的集合，每一条记录都包含任意数目的顶点和边。对输入的划分和对整个图的划分是正交的，通常都是基于文件边界进行划分。如果一个 worker 加载的顶点刚好是这个 worker 所分配到

的那一部分，那么相应的数据结构就会被立即更新。否则，该 worker 就需要将它发送到它所应属于的那个 worker 上。当所有的输入都被 load 完成后，所有的顶点将被标记为 active 状态。

4. Master 给每个 worker 发指令，让其运行一个超步，worker 轮询在其之上的顶点，会为每个 partition 启动一个线程。调用每个 active 顶点的 Compute()函数，传递给它从上一次超步发送来的消息。消息是被异步发送的，这是为了使得计算和通信可以并行，以及进行 batching，但是，消息的发送会在本超步结束前完成。当一个 worker 完成了其所有的工作后，会通知 master，并告知当前该 worker 上在下一个超步中将还有多少 active 节点。不断重复该步骤，只要有顶点还处在 active 状态，或者还有消息在传输。

5. 计算结束后，master 会给所有的 worker 发指令，让它保存它那一部分的计算结果。

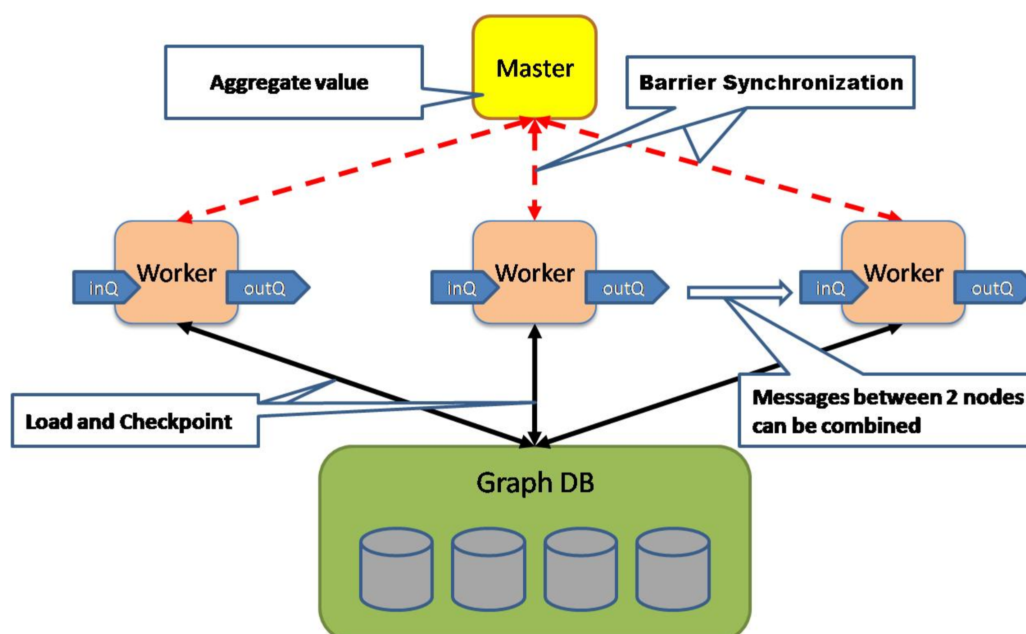


图 9-9 Pregel 的执行过程图

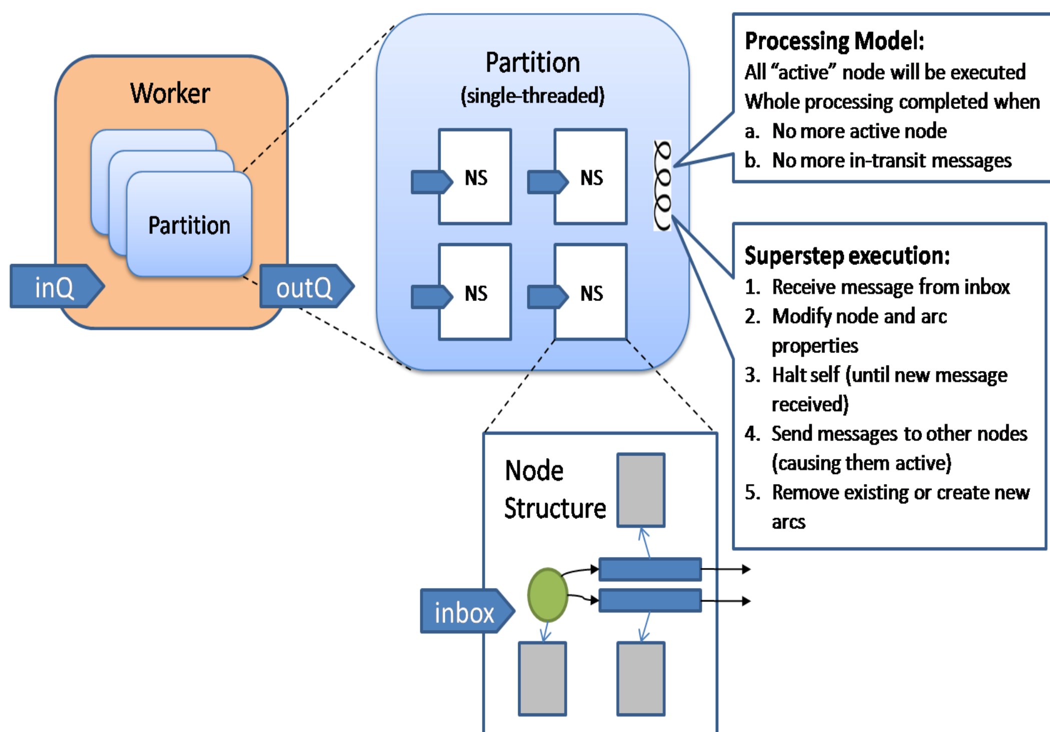


图 9-10 Worker 结构图

## 9.5.2 容错性

容错是通过 checkpointing 来实现的。在每个超步的开始阶段，master 命令 worker 让它保存它上面的 partitions 的状态到持久存储设备，包括顶点值、边值以及接收到的消息。Master 自己也会保存 aggregator 的值。

worker 的失效是通过 master 发给它的周期性的 ping 消息来检测的。如果一个 worker 在特定的时间间隔内没有收到 ping 消息，该 worker 进程会终止。如果 master 在一定时间内没有收到 worker 的反馈，就会将该 worker 进程标记为失败。

当一个或多个 worker 发生故障，被分配到这些 worker 的 partitions 的当前状态信息就丢失了。Master 重新分配图的 partition 到当前可用的 worker 集合上，所有的 partition 会从最近的某超步 S 开始时写出的 checkpoint 中，重新加载状态信息。该超步可能比在失败的 worker 上最后运行的超步 S'要早好几个阶段，此时，失去的几个超步将需要被重新执行。Pregel 对 checkpoint 频率的选择，是基于某个故障模型的平均时间的，从而平衡 checkpoint 的开销和恢复执行的开销。

为了改进恢复执行的开销和延迟， Confined recovery 已经在开发中。它们会首先通过 checkpoint 进行恢复，然后，系统会通过回放来自正常的 partitions 的记入日志的消息以及恢

复过来的 partitions 重新生成的消息，更新状态到 S’阶段。这种方式通过只对丢失的 partitions 进行重新计算，节省了在恢复时消耗的计算资源，同时，由于每个 worker 只需要恢复很少的 partitions，减少了恢复时的延迟。对发送出去的消息进行保存，会产生一定的开销，但是，通常机器上的磁盘带宽不会让这种 IO 操作成为瓶颈。

Confined recovery 要求用户算法是确定性的，以避免原始执行过程中所保存下的消息与恢复时产生的新消息并存情况下带来的不一致。随机化算法可以通过基于超步和 partition 产生一个伪随机数生成器来使之确定化。非确定性算法需要关闭 Confined recovery 而使用老的恢复机制。

## 9.5.3 Worker

一个 worker 机器会在内存中维护分配到其之上的 graph partition 的状态。从概念上来讲，可以简单地看作是一个从顶点 ID 到顶点状态的 Map，其中，顶点状态包括如下信息：该顶点的当前值，一个以该顶点为起点的出射边(包括目标顶点 ID，边本身的值)列表，一个保存了接收到的消息的队列，以及一个记录当前是否 active 的标志位。该 worker 在每个超步中，会循环遍历所有顶点，并调用每个顶点的 Compute()函数，传给该函数顶点的当前值，一个接收到的消息的迭代器和一个出射边的迭代器。这里没有对入射边的访问，原因是每一条入射边其实都是其源顶点的所有出射边的一部分，通常在另外的机器上。

出于性能的考虑，标志顶点是否为 active 的标志位，是和输入消息队列分开保存的。另外，只保存了一份顶点值和边值，但有两份顶点 active flag 和输入消息队列存在，一份是用于当前超步，另一个用于下一个超步。当一个 worker 在进行超步 S 的顶点处理时，同时还会有另外一个线程负责接收来自同一个超步的其他 worker 的消息。由于顶点当前需要的是 S-1 超步的消息，那么对超步 S 和超步 S+1 的消息就必须分开保存。类似地，顶点 V 接收到了消息，表示 V 将会在下一个超步中处于 active，而不是当前这一次。

当 Compute()请求发送一个消息到其他顶点时，worker 首先确认目标顶点是属于远程的 worker 机器，还是当前 worker。如果是在远程的 worker 机器上，那么消息就会被缓存，当缓存大小达到一个阈值，最大的那些缓存数据将会被异步地 flush 出去，作为单独的一个网络消息传输到目标 worker。如果是在当前 worker，那么就可以做相应的优化：消息就会直接被放到目标顶点的输入消息队列中。

如果用户提供了 Combiner，那么在消息被加入到输出队列或者到达输入队列时，会执

行 combiner 函数。后一种情况并不会节省网络开销，但是会节省用于消息存储的空间。

## 9.5.4 Master

Master 主要负责 worker 之间的工作协调，每一个 worker 在其注册到 master 的时候会被分配一个唯一的 ID。Master 内部维护着一个当前活动的 worker 列表，该列表中就包括每个 worker 的 ID 和地址信息，以及哪些 worker 被分配到了整个图的哪一部分。Master 中保存这些信息的数据结构大小，与 partitions 的个数相关，与图中的顶点和边的数目无关。因此，虽然只有一台 master，也足够用来协调对一个非常大的图的计算工作。

绝大部分的 master 的工作，包括输入 、输出、计算、保存以及从 checkpoint 中恢复，都将会在一个叫做 barriers 的地方终止。Master 在每一次操作时都会发送相同的指令到所有的活着的 worker，然后等待来自每个 worker 的响应。如果任何一个 worker 失败了，master 便进入恢复模式。如果 barrier 同步成功，master 便会进入下一个处理阶段，例如 master 增加超步的 index，并进入下一个超步的执行。

Master 同时还保存着整个计算过程以及整个 graph 的状态的统计数据，如图的总大小，关于出度分布的柱状图，处于 active 状态的顶点个数，在当前超步的时间信息和消息流量，以及所有用户自定义 aggregators 的值等。为方便用户监控，Master 在内部运行了一个 HTTP 服务器来显示这些信息。

## 9.5.5 Aggregators

每个 Aggregator 会通过对一组 value 值集合应用 aggregation 函数计算出一个全局值。每一个 worker 都保存了一个 aggregators 的实例集，由 type name 和实例名称来标识。当一个 worker 对 graph 的某一个 partition 执行一个超步时，worker 会 combine 所有的提供给本地的那个 aggregator 实例的值到一个 local value：即利用一个 aggregator 对当前 partition 中包含的所有顶点值进行局部归约。在超步结束时，所有 workers 会将所有包含局部归约值的 aggregators 的值进行最后的汇总，并汇报给 master。这个过程是由所有 worker 构造出一棵归约树而不是顺序地通过流水线的方式来归约，这样做的原因是为了并行化归约时 CPU 的使用。在下一个超步开始时，master 就会将 aggregators 的全局值发送给每一个 worker。

# 9.6 Pregel 的应用实例

## 9.6.1 最短路径

最短路径问题是图论中最有名的问题之一了，同时具有广泛的应用。该问题有几个形式：（1）单源最短路径，是指要找出从某个源顶点到其他所有顶点的最短路径；（2）s-t 最短路径，是指要找出给定源顶点 s 和目标顶点 t 间的最短路径，这个问题具有广泛的实验应用，比如寻找驾驶路线，并引起了广泛关注，同时它也是相对简单的；（3）全局最短路径，对于大规模的图对象来说，通常都不太实际，因为它的空间复杂度是 O(V*V)的。为了简化起见，我们这里以非常适于用 Pregel 解决的单源最短路径为例，实现代码如下：

```
class ShortestPathVertex
      : public Vertex<int, int, int> {
   void Compute(MessageIterator* msgs) {
         int mindist = IsSource(vertex_id()) ? 0 : INF;
         for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
         if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for (; !iter.Done(); iter.Next())
                  SendMessageTo(iter.Target(),
                                       mindist + iter.GetValue());
     }
     VoteToHalt();
   }
};
```

在该算法中，我们假设与顶点关联的那个值被初始化为 INF。在每个超步中，每个顶点会首先接收到来自邻居传送过来的消息，该消息包含更新过的、从源顶点到该顶点的潜在的最短距离。如果这些更新里的最小值小于该顶点当前关联值，那么顶点就会更新这个值，并

发送消息给它的邻居。在第一个超步中，只有源顶点会更新它的关联值，然后发送消息给它的直接邻居。然后这些邻居会更新它们的关联值，然后继续发送消息给它们的邻居，如此循环往复。当没有更新再发生的时候，算法就结束，之后所有顶点的关联值就是从源顶点到它的最短距离，若值为 INF 表示该顶点不可达。如果所有的边权重都是非负的，就可以保证该过程肯定会结束。

该算法中的消息保存都是潜在的最小距离。由于接收顶点实际上只关注最小值，因此该算法是可以通过 combiner 进行优化的，combiner 实现代码如下所示，它可以大大减少 worker 间的消息量，以及在执行下一个超步前所需要缓存的数据量。

```
class MinIntCombiner : public Combiner<int> {
    virtual void Combine(MessageIterator* msgs) {
        int mindist = INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        Output("combined_source", mindist);
    }
};
```

与其他类似的串行算法（比如 Dijkstra 或者 Bellman-Ford）相比，该算法需要更多的比较次数，但是，它可以用来解决对于单机版实现很难解决的那个规模上的最短路径问题。还有一些更高级的并行算法，比如 Thorup 和△ -stepping 算法，这些高级算法也可以在 Pregel 系统中实现。但是，上述代码的实现，由于其比较简单同时性能也还可以接受，对于那些普通用户来说也还是很具有吸引力的。

## 9.6.2 二分匹配

二分匹配算法的输入由两个不同的顶点集合组成，所有边的两个顶点分别位于两个集合中，输出是边的一个子集，它们之间没有公共顶点。极大匹配(Maximal Matching)是指在当前已完成的匹配下，无法再通过增加未完成匹配的边的方式来增加匹配的边数。Pregel 实现了一个随机化的极大匹配算法以及一个最大权匹配算法；下面是随机化的极大匹配算法。

```
Class BipartiteMatchingVertex
        : public Vertex<tuple<position, int>, void, boolean> {
    public:
        virtual void Compute(MessageIterator* msgs) {
            switch (superstep() % 4) {
                case 0: if (GetValue().first == 'L') {
                        SendMessageToAllNeighbors(1);
                        VoteToHalt();
                    }
                case 1: if (GetValue().first == 'R') {
                    Rand myRand = new Rand(Time());
                    for ( ; !msgs->Done(); msgs->Next()) {
                        if (myRand.nextBoolean()) {
                            SendMessageTo(msgs->Source, 1);
                            break;
                        }
                    }
                    VoteToHalt(); }
            case 2:
                    if (GetValue().first == 'L') {
                        Rand myRand = new Rand(Time());
                        for ( ; !msgs->Done(); msgs->Next) {
                            if (myRand.nextBoolean()) {
                                *MutableValue().second = msgs->Source());
                                SendMessageTo(msgs->Source(), 1);
                                break;
                            }
                        }
                        VoteToHalt(); }
```

```
            case 3:
              if (GetValue().first == 'R') {
                msgs->Next();
                *MutableValue().second = msgs->Source();
              }
              VoteToHalt();
      }}};
```

在该算法的 Pregel 实现中，顶点的关联值是由两个值组成的元组(tuple)：一个是用于标识该顶点所处集合(L or R)的 flag，一个是跟它所匹配的顶点名称。边的关联值类型为 void，消息的类型为 boolean。该算法是由四个阶段组成的多个循环组成（如图 9-11 所示），用来标识当前所处阶段的 index 可以通过用当前超步的 index mod 4 得到。

在循环的阶段 0，左边集合中那些还未被匹配的顶点会发送消息给它的每个邻居请求匹配，然后会无条件地 VoteToHalt。如果它没有发送消息(可能是因为它已经找到了匹配，或者没有出射边)，或者是所有的消息接收者都已经被匹配，该顶点就不会再变为 active 状态。

在循环的阶段 1，右边集合中那些还未被匹配的顶点随机选择它接收到的消息中的其中一个，并发送消息表示接受该请求，然后给其他请求者发送拒绝消息。然后，它也无条件地 VoteToHalt。

在循环的阶段 2，左边集合中那些还未被匹配的顶点选择它所收到右边集合发送过来的接受请求中的其中一个，并发送一个确认消息。左边集合中那些已经匹配好的顶点永远都不会执行这个阶段，因为它们不会在阶段 0 发送任何消息。

在循环的阶段 3，右边集合中还未被匹配的顶点最多会收到一个确认消息。它会通知匹配顶点，然后无条件地 VoteToHalt，它的工作已经完成。

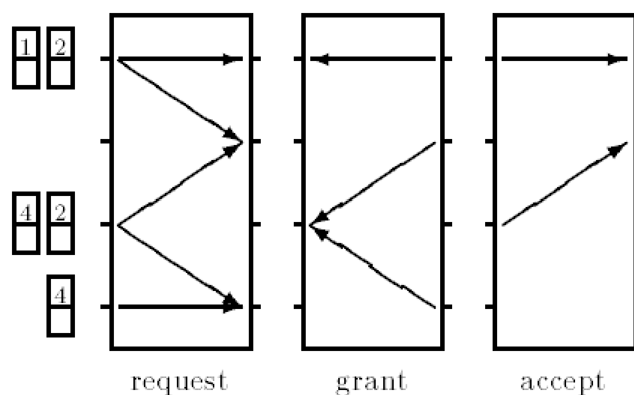


图 9-11 循环的四个阶段

# 9.7 改进的图计算模型

## 9.7.1 Pregel 的不足之处

作为第一个通用的大规模图处理系统，Pregel 已经为分布式图处理迈进了不小的一步，这点不容置疑，但是，Pregel 在一些地方也不尽如人意，具体如下：

1.在图的划分上，采用的是简单的 hash 方式，这样固然能够满足负载均衡，但是 hash 方式并不能根据图的连通特性进行划分，导致超步之间的消息传递开销可能会是影响性能的最大隐患。

2.简单的 checkpoint 机制只能向后式地将状态恢复到当前 S 超步的几个超步之前，要到达 S 还需要重复计算，这其实也浪费了很多时间，因此，如何设计 checkpoint，使得只需重复计算故障 worker 的 partition 的计算，从而节省计算，甚至可以通过 checkpoint 直接到达故障发生前一超步 S，也是一个很需要研究的地方。

3.BSP 模型本身有其局限性，整体同步并行对于计算快的 worker 长期等待的问题，仍然无法解决。

4.由于 Pregel 目前的计算状态都是常驻内存的，对于规模继续增大的图处理可能会导致内存不足，如何解决尚待研究。

## 9.7.2 PowerGraph

PowerGraph 将基于 vertex 的图计算抽象成一个通用的计算模型：GAS 模型（代码实现如下），分为三个阶段：Gather、Apply 和 Scatter。

1. Gather 阶段：用户自定义一个 sum 操作，用于各个顶点，将顶点的相邻顶点和对应边收集起来；

2. Apply 阶段：各个顶点利用上一阶段的 sum 值进行计算，并更新原始值；

3. Scatter 阶段：利用第二阶段的计算结果，更新顶点相连的边的值。

```
// gather_nbrs: IN_NBRS
gather(Du, D(u,v), Dv):
    return Dv.rank / #outNbrs(v)
```

```
sum(a, b): return a + b
apply(Du, acc):
    rnew = 0.15 + 0.85 * acc
    Du.delta = (rnew - Du.rank)/
                #outNbrs(u)
    Du.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(Du,D(u,v),Dv):
    if(|Du.delta|>) Activate(v)
return delta
```

由于顶点计算会频繁调用 Gather 阶段操作，而大多数相邻的顶点的值其实并不会变化，为了减少计算量，PowerGraph 提供了一种 Cache 机制，上面显示了 PowerGraph 机制下 Page Rank 计算的过程伪代码。

PowerGraph 提出了一种均衡图划分方案，减少计算中通信量的同时保证负载均衡。与 Pregel 和 GraphLab 均采用的 hash 随机分配方案不同，它提出了一种均衡 p-路顶点切割（vertex-cut）分区方案。根据图的整体分布概率密度函数计算顶点切割的期望值：

$$\mathbb{E}\left[\left(1-\frac{1}{p}\right)^{\mathbf{D}[v]}\right]=\frac{1}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(1-\frac{1}{p}\right)^{d}d^{-\alpha}.$$

根据该期望值指导对顶点进行切割，并修改了传统的通信过程，具体如图 9-12 所示。

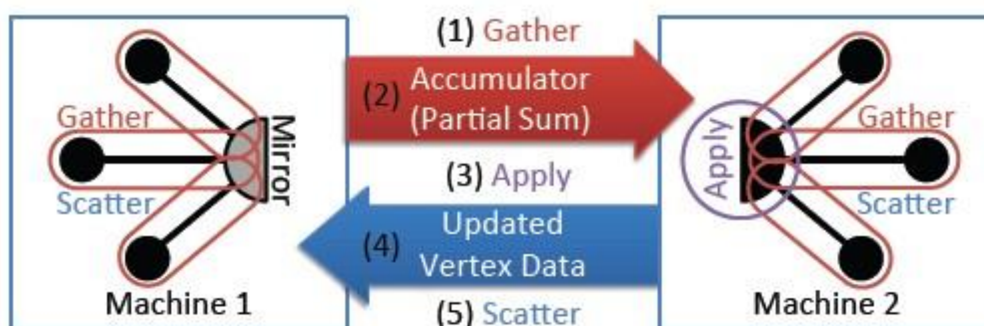


图 9-12 通信过程图

实验时，PowerGraph 按同步方式不同分别实现了三种版本（全局同步、全局异步和可串行化异步），具体如下：

- **全局同步**：与 Pregel 类似，超步之间设置全局同步点，用以同步对所有边以及顶点的

修改；

- **全局异步**：类似 GraphLab，所有 Apply 阶段和 Scatter 阶段对边或顶点的修改立即更新到图中；
- **可串行化异步**：全局异步会使得算法设计和调试变得很复杂，对某些算法效率可能比全局同步还差，因此有全局异步加可串行化结合的方式。

在差错控制上，依靠 checkpoint 的实现，采用 GraphLab 中使用的 Chandy-Lamport 快照算法。

通过将图计算模型进行抽象，设计实现均衡的图划分方案，对比三种不同方式下的系统实现，并实现了差错控制。

# 本章小结

本章首先简单介绍了图计算中的问题，介绍了处理图计算的两种软件，接着介绍了 Pregel 和 MapReduce 两种框架在 PageRank 算法中的实现，主要为了说明 Pregel 处理图计算问题的优势。然后主要重点介绍了 Pregel 图计算模型，它的 C++ API、执行过程和算法实现。最后介绍了一种对 Pregel 改进的图计算框架 PowerGraph。

# 参考文献

[1] Pregel——大规模图处理系统 http://www.cnblogs.com/panfeng412/archive/2011/10/28/2227195.html

[2]PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs(OSDI'12) http://wuyanzan60688.blog.163.com/blog/static/127776163201312863021180/

[3] PageRank 算法在 Pregel 和 MapReduce 两种计算模型中的思路. http://wuyanzan60688.blog.163.com/blog/static/1277761632012111043525435/

[4] 被冷落的大数据热点：图谱分析. http://www.ctocio.com/ccnews/12340.html

[5] 论文 Pregel: A System for Large-Scale Graph Processing.

# 附录 1:任课教师介绍

**林子雨**(1978－)，男，博士，厦门大学计算机科学系助理教授，主要研究领域为数据库，数据仓库，数据挖掘。
主讲课程：《大数据技术基础》
办公地点：厦门大学海韵园科研 2 号楼
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu