厦门大学计算机科学系研究生课程

《大数据技术基础》

第9章 图计算

（2013年新版）

林子雨

厦门大学计算机科学系

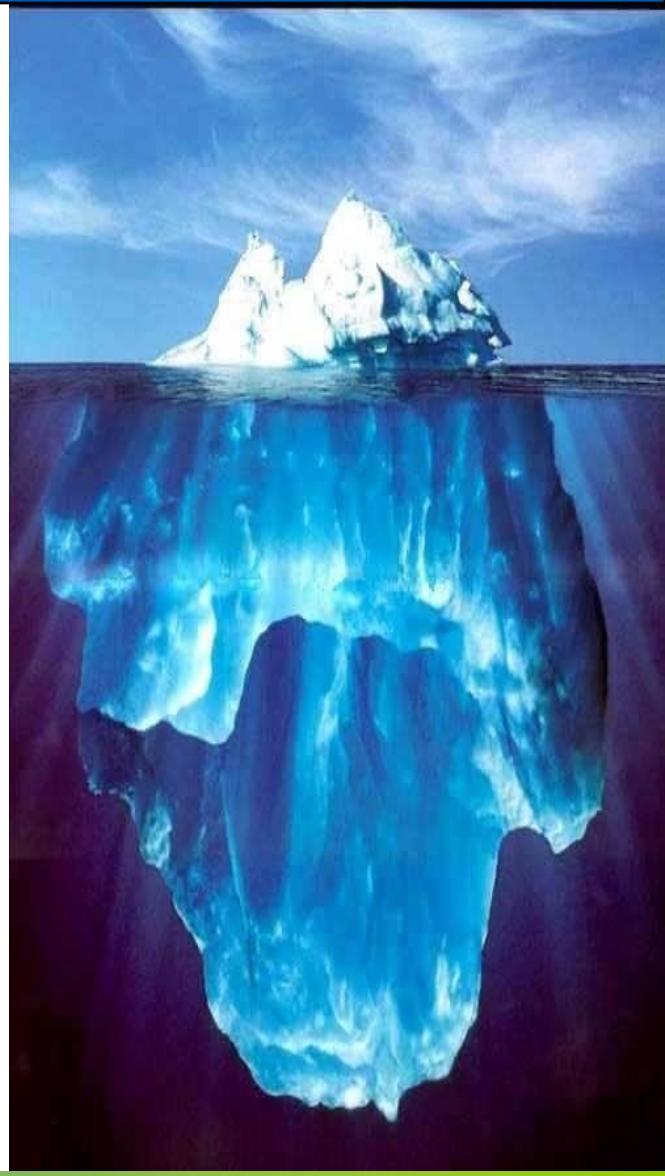
E-mail: ziyulin@xmu.edu.cn

主页：http://www.cs.xmu.edu.cn/linziyu

课程提要

- 图计算简介
- 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
- 参考资料

本讲义PPT存在配套教材，由林子雨通过大量阅读、收集、整理各种资料后编写而成
下载配套教材请访问《大数据技术基础》2013班级网站：http://dblab.xmu.edu.cn/node/423



图计算中的问题

- 大型图（像社交网络和网络图等）常常作为现在系统计算需要的一部分。现在存在许多图计算问题像最短路径、集群、网页排名、最小切割、连通分支等等，但还没有一个可扩展的通用系统来解决这些问题。
- 解决这些问题的算法的特点：它们常常表现为比较差的内存访问局部性、针对单个顶点的处理工作过少、以及计算过程中伴随着的并行度的改变等问题。
- 可能的解决方法：
  - 为特定的图应用定制相应的分布式实现
  - 基于现有的分布式计算平台
  - 使用单机的图算法库

    ——如BGL，LEAD，NetworkX，JDSL，Standford ，GraphBase，FGL等
  - 使用已有的并行图计算系统

    ——如Parallel BGL，CGMgraph等

图计算的两种软件

目前通用的图处理软件主要包括两种。一种主要基于遍历算法、实时的图数据库，如 Neo4j , OrientDB , DEX , 和 InfiniteGraph .另一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 .第一种基本都基于tinkerpop的图基础框架，tinkerpop项目关系如图1所示：

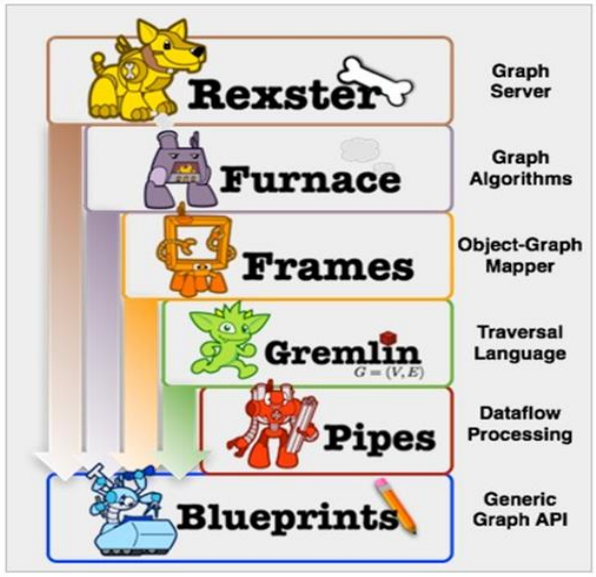


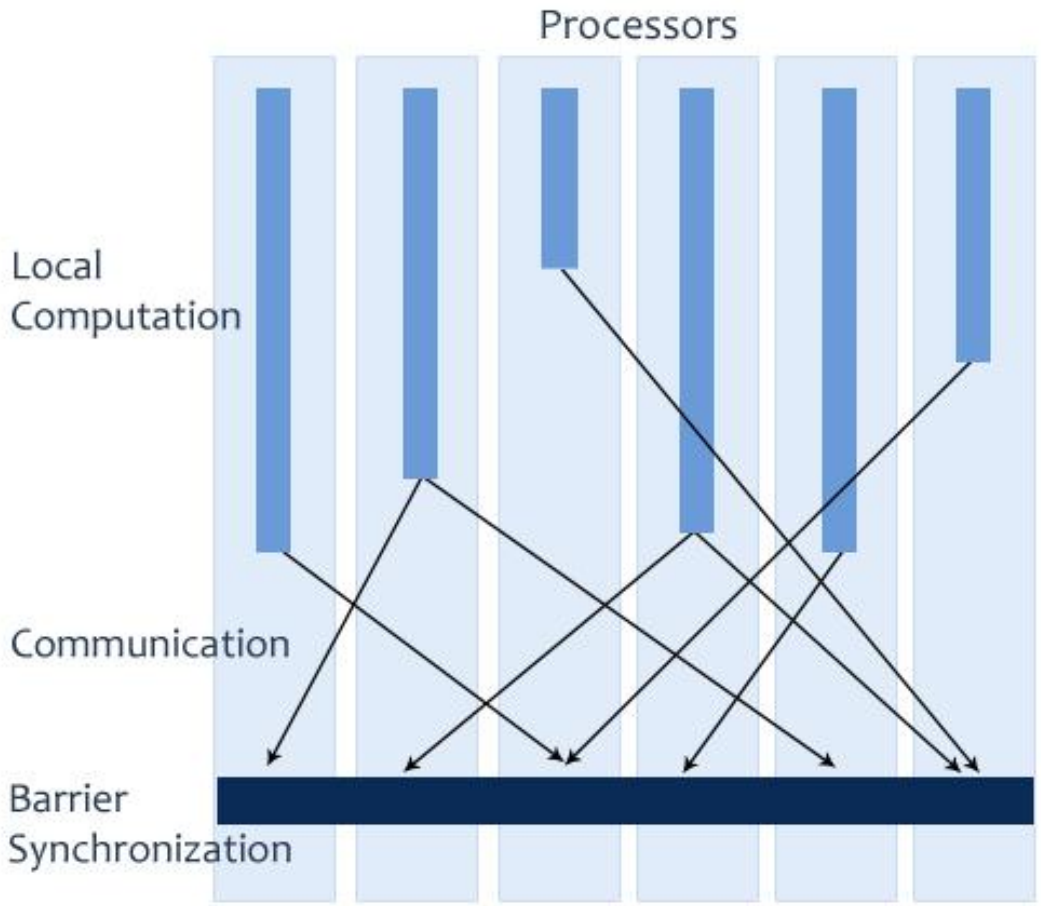
图1 thinkerpop项目框架

BSP模型

以图顶点为中心的消息传递批处理的并行引擎主要是基于BSP(Bulk Synchronous Parallel)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。一个BSP模型由大量相互关联的处理器(processor)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计算过程由一系列全局超步组成，超步就是计算中一次迭代。每个超步主要包括三个组件：

- 并发计算(Concurrent computation)：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(Communication): 处理器群相互交换数据，交换的形式：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。

Superstep



Vertical Structure of a Superstep

Pregel图计算框架

Pregel是由Google开发的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，所以需要分布式的这样一个计算环境。没有Pregel之前，可以选择用MapReduce来做，但是效率很低。下面简单介绍一下PageRank算法在Pregel和MapReduce中的实现。

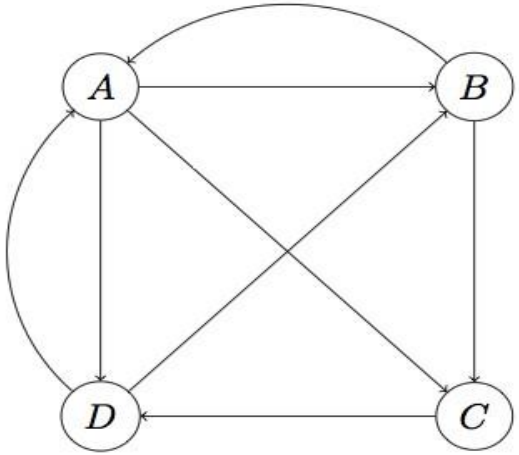
PageRank算法作为Google的网页链接排名算法，具体公式如下：

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

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N_i为第i个源链接的链出度）。

Pregel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，网络链接是一个连通图。

PageRank在Pregel中的实现



上图就是四个网页（A,B,C,D）互相链入链出组成的联通图。根据Pregel的计算模型，将计算定义到顶点（vertex）即A,B,C,D上来，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vertex对其它vertex的PR贡献传递给目标vertex

每一个计算单元包含一个成员函数：

- Compute：该函数定义了vertex上的运算，包括该vertex的PR值计算，以及从该vertex发送消息到其链出vertex

PageRank在Pregel中的实现

```
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();
                }
        }
};
```

PageRank在Pregel中的实现

- Pregel的执行包含PageRankVertex类，它继承了Vertex类。
- 该类顶点值的类型是double，用来存储暂定的PageRank，消息类型也是double，用来传递PageRank的部分。
- 图在第0个超步中被初始化，所以它的每个顶点值为1.0。
- 在每个超步中，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。
- 为了收敛，可以设置一个超步数量的限制或用aggregators来检查是否满足收敛条件

PageRank在MapReduce中的实现

MapReduce也是Google提出的一种计算模型，它是为全量计算而设计。它实现MapReduce需要以下三个阶段：

- 阶段1：解析网页
  - Map task把（URL，page content）对映射为（URL，（PR_{init}，list-of-urls））
    - ✓ PR_{init}是URL的“seed” PageRank。
    - ✓ list-of-urls包含通过URL指向的所有页。
  - Reduce task只是恒等函数。
- 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|）。
    - ✓ 输出（URL，url_list）通过迭代器来获取列表的指向。
  - Reduce task 获得（URL，url_list）和很多（URL，var）值对
    - ✓ 汇总vals，乘上d(0.85)。
    - ✓ 输出（URL，（new_rank，url_list））。
- 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。

PageRank在MapReduce中的实现

- 下面是第二阶段，把网页链接映射到key-value对的伪代码：

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
```

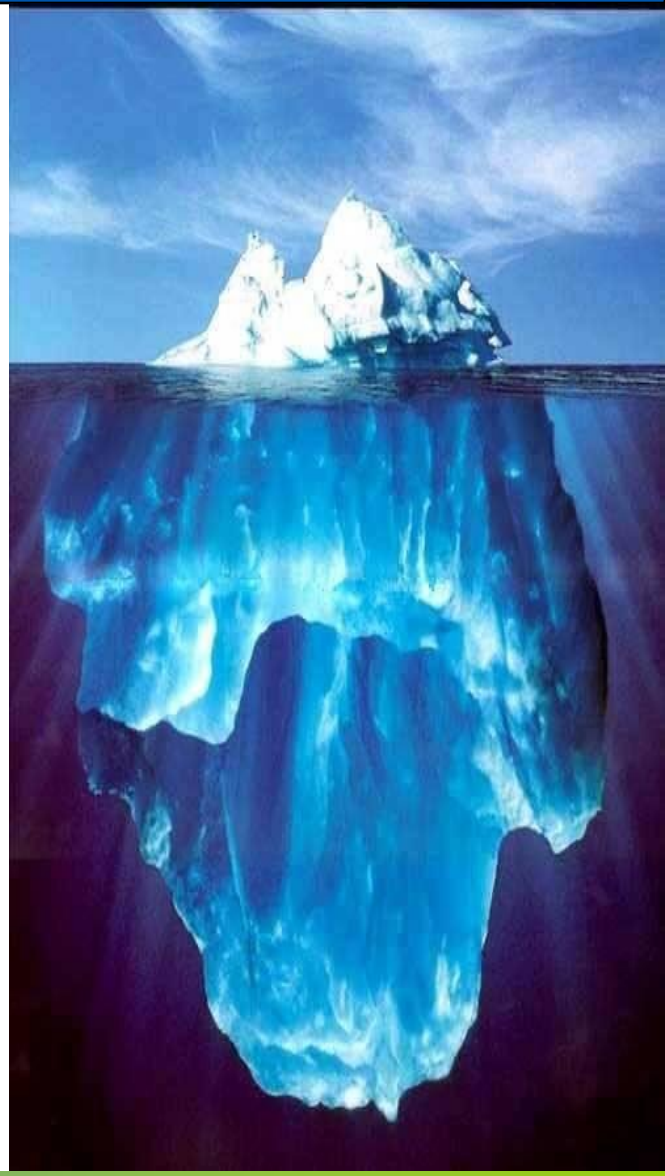
PageRank在两种模型中实现的总结

总结：

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。PageRank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，这样就需要许多的通信和随之而来的序列化和反序列化的开销。另外，这一连串的MapReduce作业各执行阶段需要的协同工作也增加了编程复杂度，而Pregel使用超步简化了这个过程。

课程提要

- 图计算简介
- 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
- 参考资料



Pregel的计算

➢一个典型的Pregel计算过程如下：读取输入初始化该图，当图被初始化好后，运行一系列的超步直到整个计算结束，这些超步之间通过一些全局的同步点分隔开，输出结果结束计算。

➢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。函数可以：

- 读取前一个超步(S-1)中发送给V的消息
- 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
- 修改顶点V及其出射边的状态
- 发生拓扑变化

➢整个Pregel程序的输出是所有顶点输出的集合。

顶点：

➢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。

边：

➢每一条有向边都和其源顶点关联，还记录了其目标顶点的标识符。

➢每一条有向边拥有一个可修改的用户自定义值。

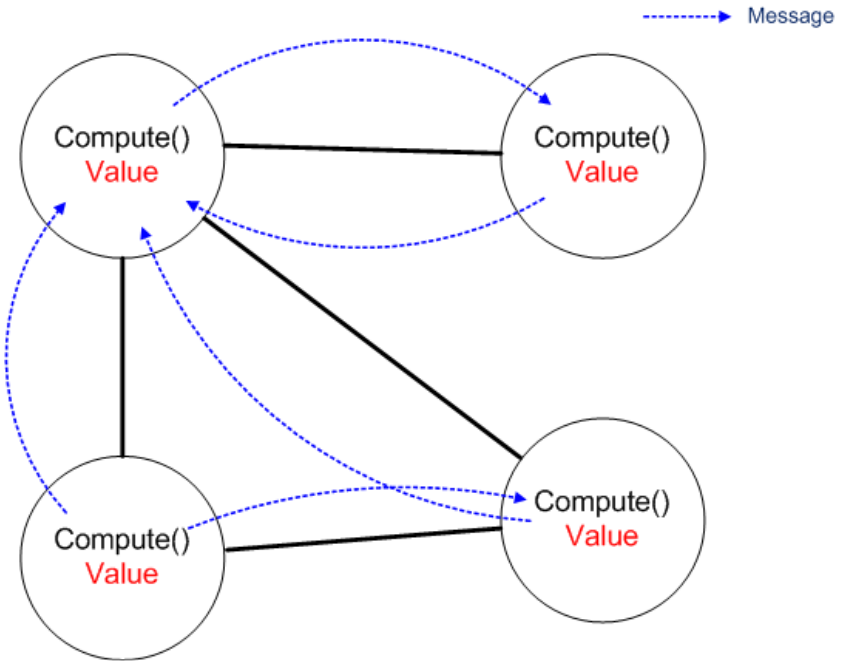
Pregel计算模型的进程

- 在第0个超步，所有顶点都处于active状态
- 只有active顶点参与对应超步中的计算
  - 顶点通过将其自身的status设置成“halt”来进入inactive状态
  - inactive顶点收到其它顶点传送的消息被唤醒进入active状态
- 整个计算在所有顶点都达到“inactive”状态，并且没有message在传送的时候宣告结束。

Pregel的消息传递模型

➢ 计算模型是一种纯消息传递模型，忽略远程数据读取和其他共享内存的方式，有两个原因。

- 第一，消息传递模型足够表达所有图算法。
- 第二，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。而我们的消息传递模式通过异步的方式传输批量消息，可以减少远程读取的延迟。



Pregel的一个消息传递的例子

- 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，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。

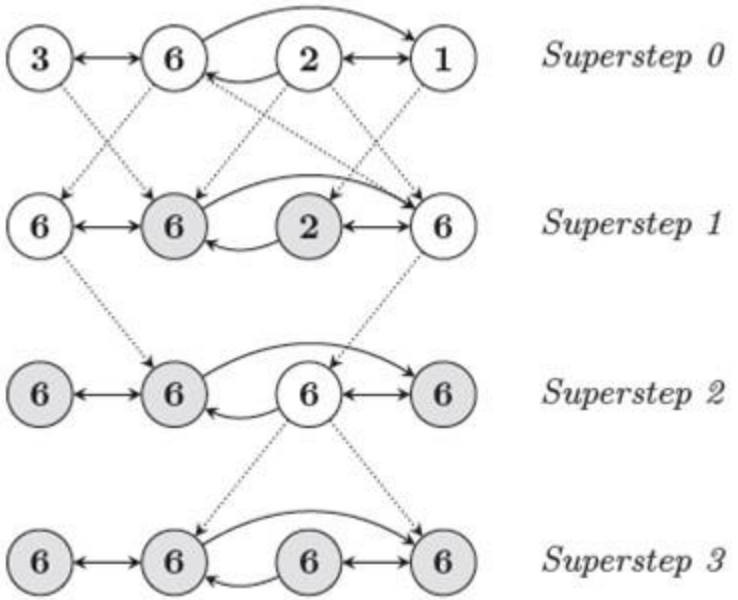
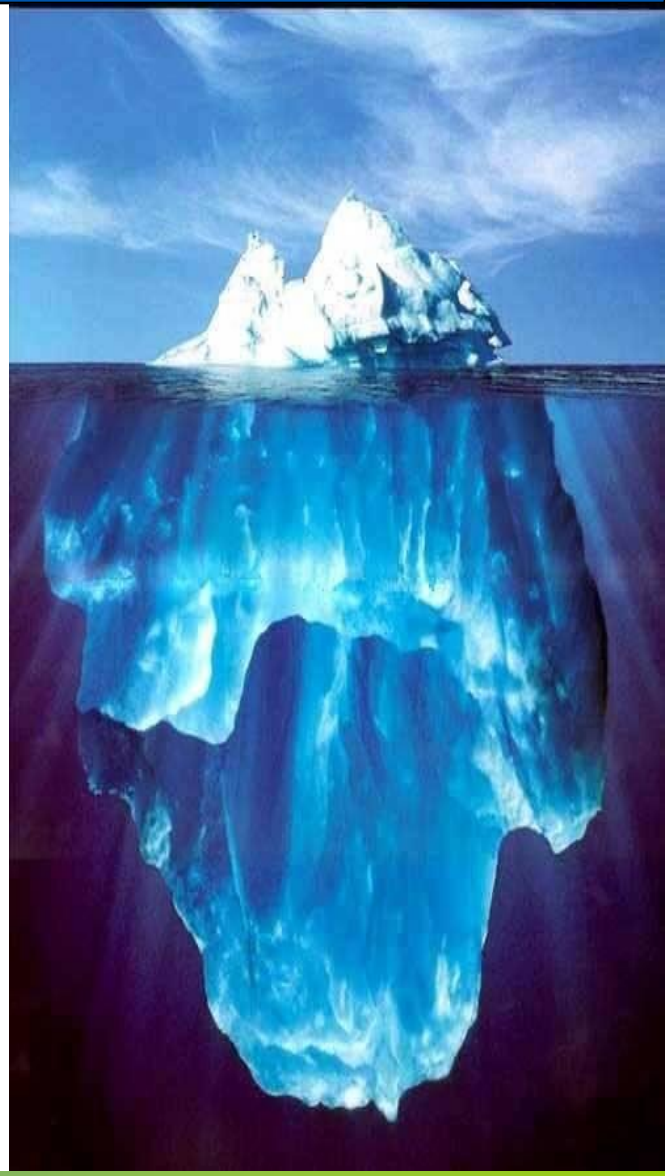


Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

课程提要

- 图计算简介
- 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
- 参考资料



Pregel 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

```
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();
};
```

- 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。
- 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。
- 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。

消息传递机制

- 顶点之间的通信是直接通过发送消息，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。
  - 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。
  - 消息可以传给任意标识符已知的顶点

combiner

- 发送消息时，尤其是当目标顶点在另外一台机器时，会产生一些开销。某些情况可以用combiner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combine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。

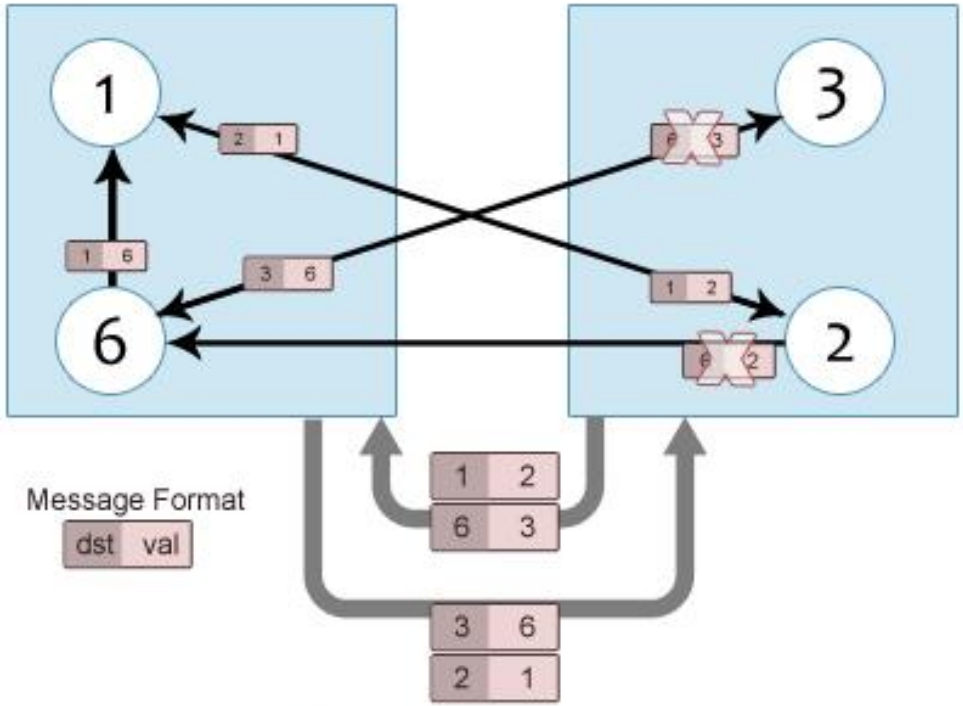
- Combiners在默认情况下并没有被开启，而用户如果想要开启Combiner的功能，可以通过重载Combine（）方法实现。框架并不会确保哪些消息会被Combine而哪些不会，也不会确保传送给Combine()的值和Combining操作的执行顺序。所以Combiner只应该对那些满足交换律和结合律的操作打开。

combiner

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）的链接，我们会向目标页面发送一个消息。
- 这里输入消息队列上的count函数可以通过一个combiner来优化性能。

在这个求最大值的例子中，一个Max combiner可以减少通信负荷。



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。
- Aggregators可以用来做统计和全局协同。
- Aggregators可以通过把Aggregator类子类化来实现。
  - 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。

• 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Reduce运算符甚至可以产生直方图。
- 在求最大值得例子中，我们我们可以通过运用一个**Max** aggregator在一个超步中完成整个程序。

topology mutation

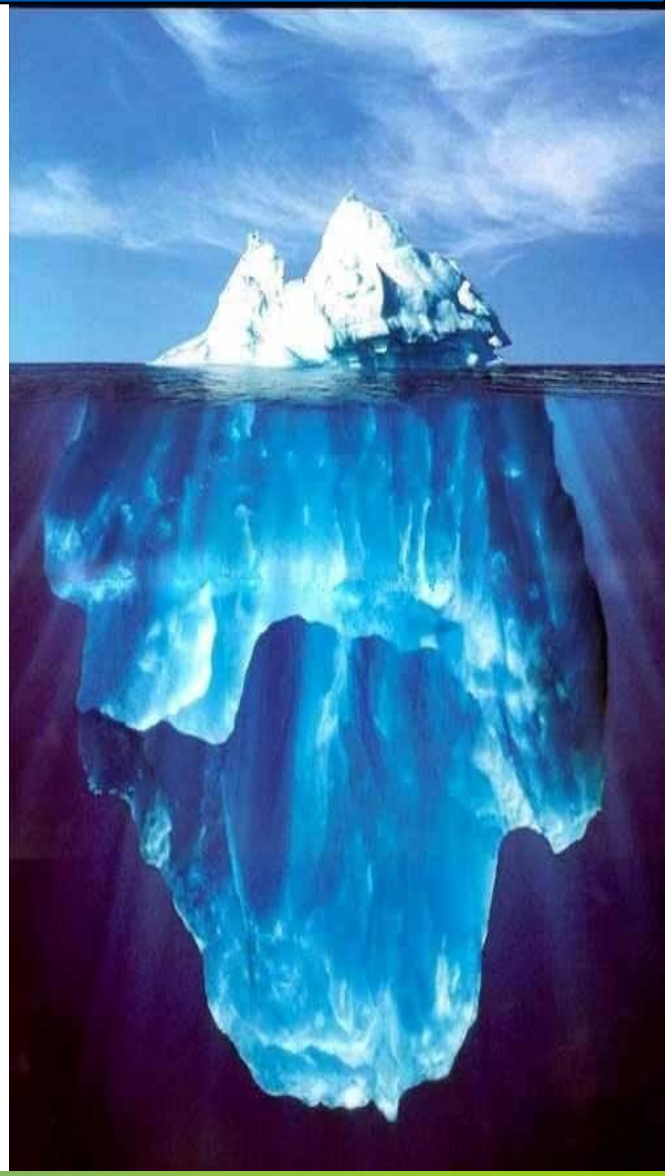
- Compute()算法也可以用来修改图的拓扑结构。
- 在请求发出后在该超步中发生拓扑变化。拓扑变化的顺序：
  - 删除操作在添加操作之前
  - 删除边操作在删除顶点操作之前
  - 添加顶点操作在添加边操作之前
- 这种局部有序性解决了很多冲突，其余的冲突由用户自定义的handlers解决。同一种handler机制将被用于解决由于多个顶点删除请求或多个边增加请求或删除请求而造成的冲突。
- Pregel的协同机制是惰性的，全局的拓扑改变在被apply之前不需要进行协调这种设计的选择是为了优化流式处理。直观来讲就是对顶点V的修改引发的冲突由V自己来处理。
- Pregel同样也支持纯local的拓扑改变，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的编程。

Input and Output

- 可以采用多种格式进行图的保存，比如可以用text文件，关系数据库，或者Bigtable中的行。
- 类似的，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。
- 用户可以通过继承Reader和Writer类来定义他们自己的读写方式。

课程提要

- 图计算简介
- 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
- 参考资料



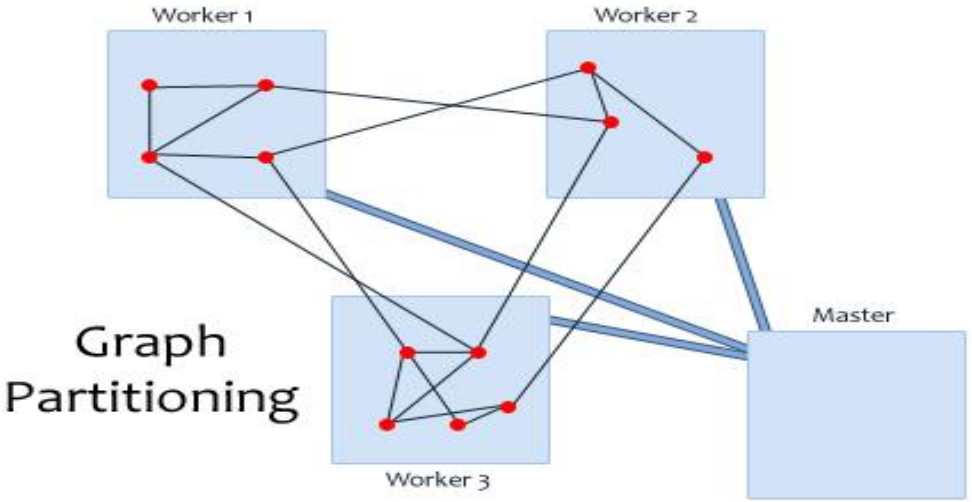
Implementation

- Pregel是为Google的集群架构而设计的。
- 应用程序通常通过一个集群管理系统执行，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。
- 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。
- Pregel library将一张图划分成许多的partitions，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。
- 默认的partition函数为hash(ID) mod N，N为所有partition总数。

接下来描述一个Pregel程序执行的几个阶段。

执行过程

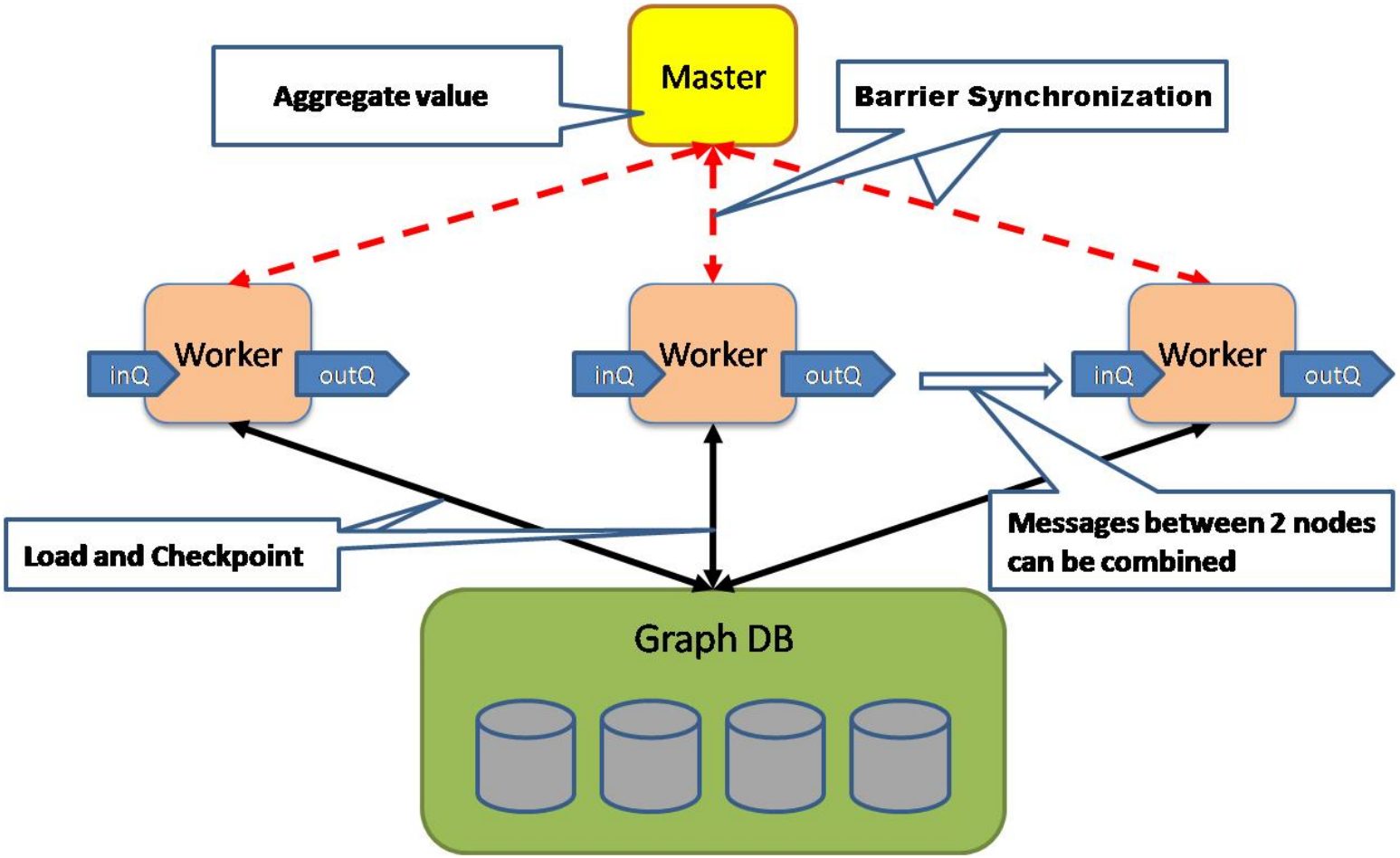
- 1.用户程序的多个copy开始在集群中的机器上执行。其中一个copy充当master
  - Master不被分配图的任意部分，它负责协调worker的活动
- 2.master将图进行分区，然后将一个或多个partition分给worker；
  - 每一个worker会在内存中维护分配到其之上的graph partition的状态。执行它的顶点上的用户定义的Compute()方法并管理来自或发给其他顶点的消息。



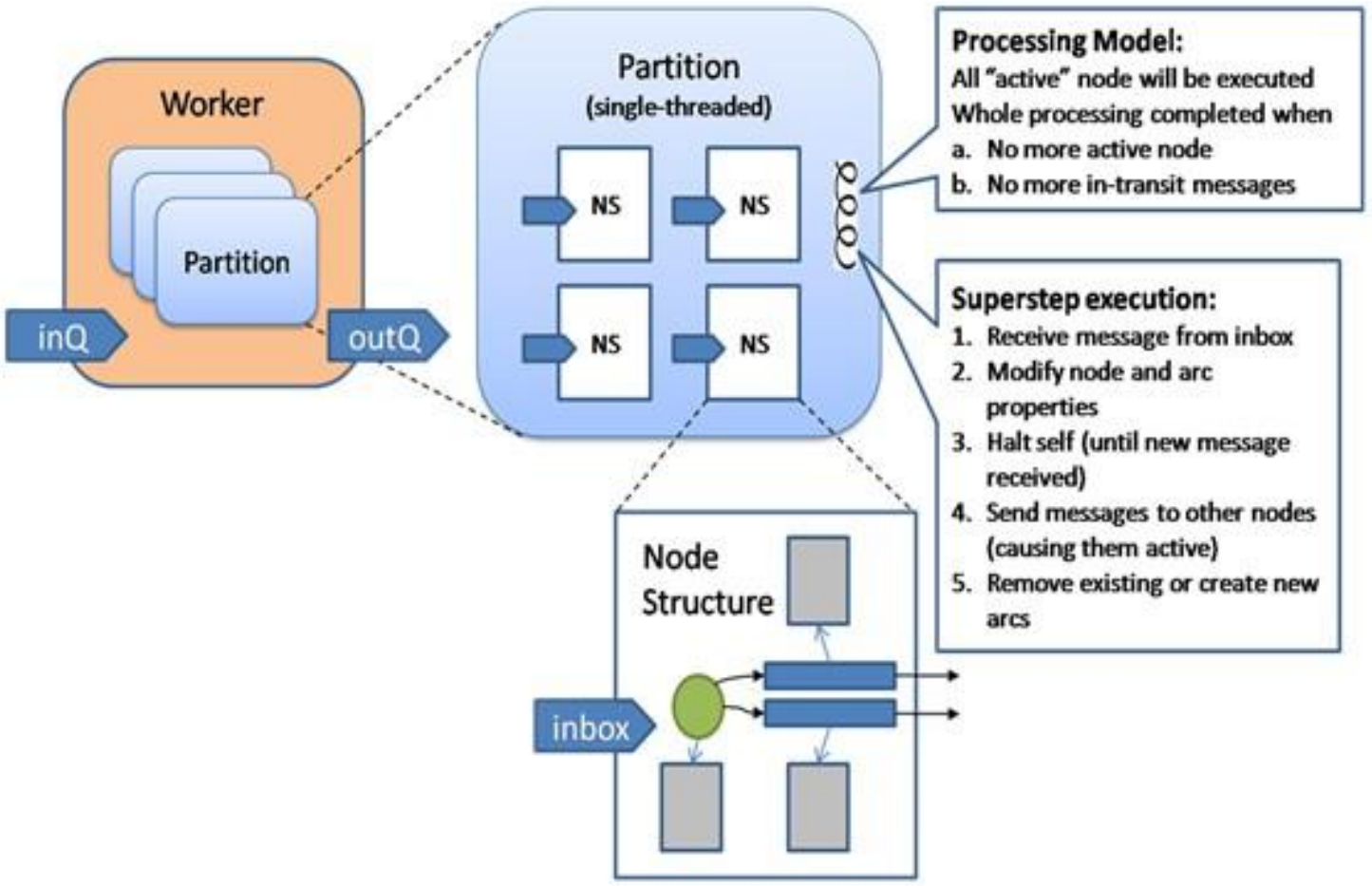
执行过程

- 3.Master为每个worker分配用户输入的一部分。
  - 输入被看做一系列的记录，每个记录包含任意数量的顶点和边。
  - 在输入完成加载后，所有的顶点被标记为active。
- 4.在一个超步中，master通知每一个worker去执行，只要存在active顶点worker一直执行，并为每一个active状态的顶点调用compute()方法。它也会传送以前的超步发送的消息。
  - 当worker完成后，它会向master作出响应，告诉master在下一个超步中active顶点的数量。
- 5.计算结束后，master会通知所有的worker保存它那部分的计算结果。

执行过程


Aggregate value
Master
Barrier Synchronization
Worker
inQ
outQ
Worker
inQ
outQ
Worker
inQ
outQ
Load and Checkpoint
Messages between 2 nodes can be combined
Graph DB

执行过程


Worker
Partition
inQ
outQ
Partition
(single-threaded)
NS
NS
NS
NS
Node
Structure
inbox
Processing Model:
All "active" node will be executed
Whole processing completed when
a. No more active node
b. No more in-transit messages
Superstep execution:
1. Receive message from inbox
2. Modify node and arc properties
3. Halt self (until new message received)
4. Send messages to other nodes (causing them active)
5. Remove existing or create new arcs

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通过ping消息检测worker是否故障
  - 当一个或多个worker出现故障时，和它们关联的分区的当前状态就会丢失。
- 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。该超步可能比在出故障的worker上最后运行的超步 S’早好几个阶段
  - 整个系统从该超步重新开始
- Confined recovery可以改进恢复执行的开销和延迟。除了基本的checkpoint，worker同时还会将其在加载图的过程中和超步中发送出去的消息写入日志。这样恢复就会被限制在丢掉的那些 partitions上。

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。
- 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。

Master

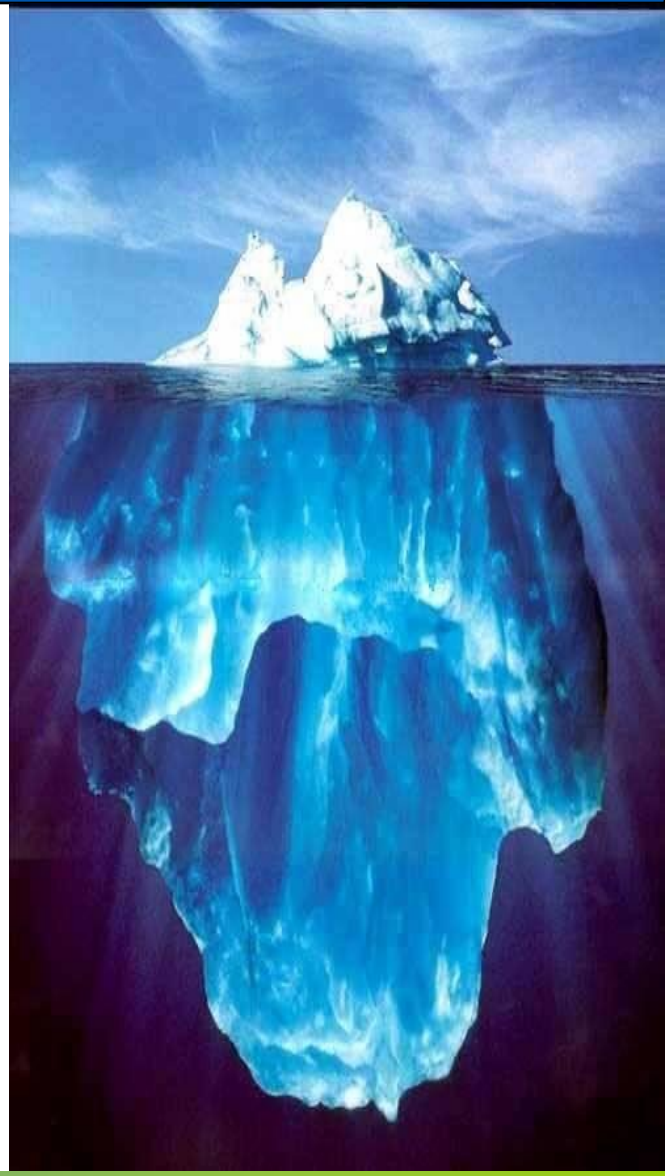
- Master主要负责的worker之间的工作协调，每一个worker在其注册到master的时候会被分配一个唯一的ID。Master内部维护着一个当前活动的worker列表，master中保存这些信息的数据结构大小与partitions的个数相关，与图中的顶点和边的数目无关。
- 绝大部分的master的工作，包括输入，输出，计算，保存以及从 checkpoint中恢复，都将会在一个叫做barriers的地方终止：
- Master同时还保存着整个计算过程以及整个graph的状态的统计数据。为方便用户监控，Master在内部运行了一个HTTP服务器来显示这些信息。

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执行一个超级步时，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规约。在超级步结束时，所有workers会将所有包含局部规约值的aggregators的值进行最后的汇总，并汇报给master。这个过程是由所有worker构造出一棵规约树而不是顺序的通过流水线的方式来规约，这样做的原因是为了并行化规约时cpu的使用。在下一个超级步开始时，master就会将aggregators的全局值发送给每一个worker。

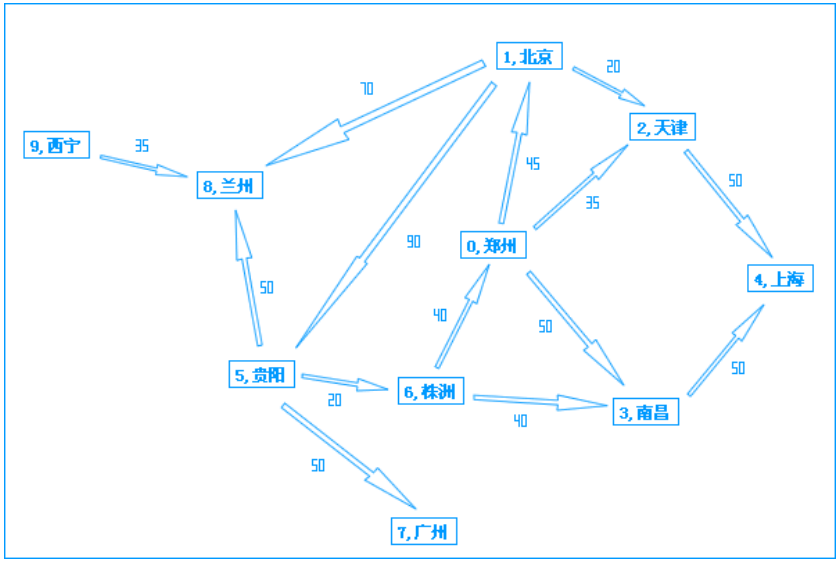
课程提要

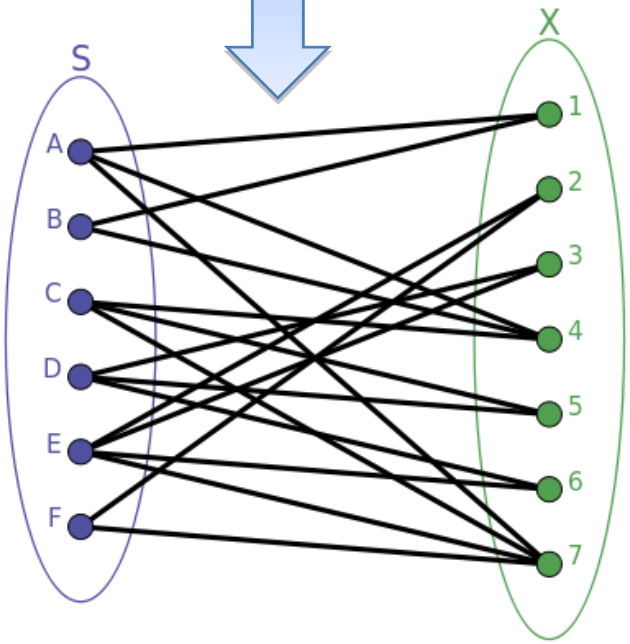
- 图计算简介
- Google Pregel图计算模型
- Pregel的C++ API
- Pregel模型的基本体系结构
- Pregel模型的应用实例
- 改进的图计算模型
- 参考资料



应用实例

- 最短路径
- 二分匹配





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：单源最短路径、s-t最短路径、全局最短路径。

```
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();
  }
};
```

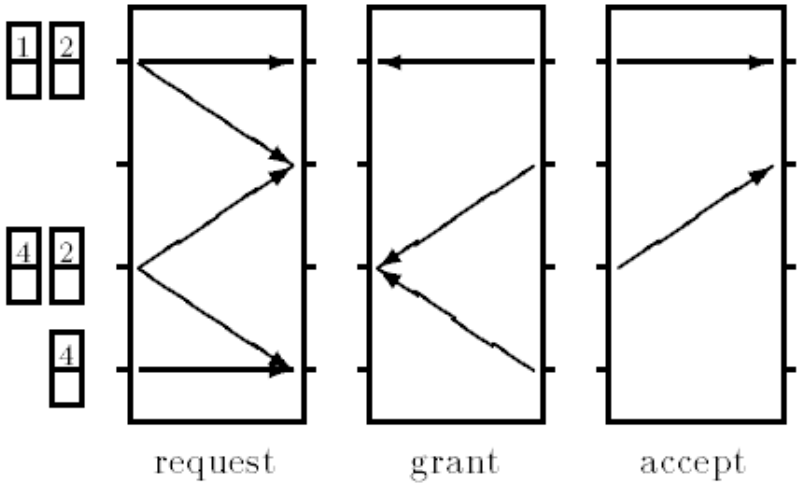
Figure 5: Single-source shortest paths.

单源最短路径

- 在算法中我们假设每个顶点的关联值被初始化为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。

二分匹配

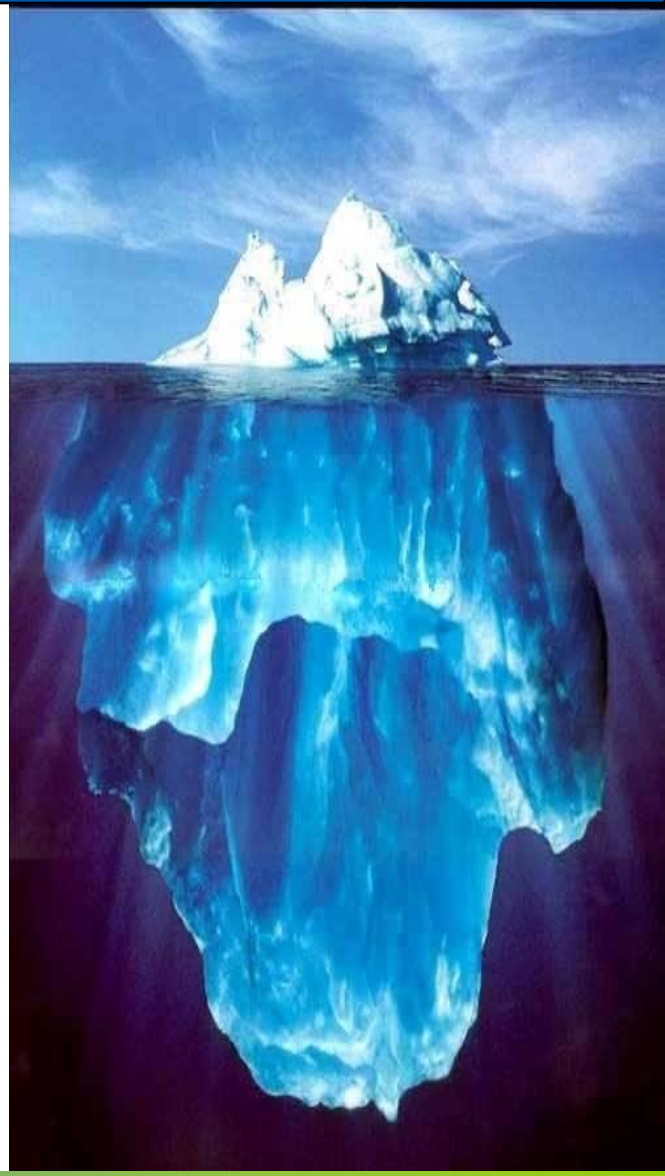
- 在循环的阶段0，左边集合中那些还未被匹配的顶点会发送消息给它的每个邻居请求匹配，然后会无条件的VoteToHalt。如果它没有发送消息，或者是所有的消息接收者都已经被匹配，该顶点就不会再变为active状态。
- 在循环的阶段1，右边集合中那些还未被匹配的顶点随机选择它接收到的消息中的其中一个，并发送消息表示接受该请求，然后给其他请求者发送拒绝消息。然后，它也无条件的VoteToHalt。
- 在循环的阶段2，左边集合中那些还未被匹配的顶点选择它所收到右边集合发送过来的接受请求中的其中一个，并发送一个确认消息。左边集合中那些已经匹配好的顶点永远都不会执行这个阶段，因为它们不会在阶段0发送任何消息。
- 最后，在阶段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，然后无条件的VoteToHalt，它的工作已经完成。



Execution of a cycle (A cycle consists of 4 supersteps)

课程提要

- 图计算简介
- 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
- 参考资料



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。

- 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。

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，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vertex，将vertex的相邻vertex和对应edge收集起来；
- 2. Apply阶段各个vertex利用上一阶段的sum值进行计算更新原始值；
- 3. Scatter阶段利用第二阶段的计算结果更新vertex相连的edge值。

由于vertex计算会频繁调用Gather阶段操作，而大多数相邻vertex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，下面是PowerGraph机制下Page Rank计算的过程伪代码。

```
// 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
```

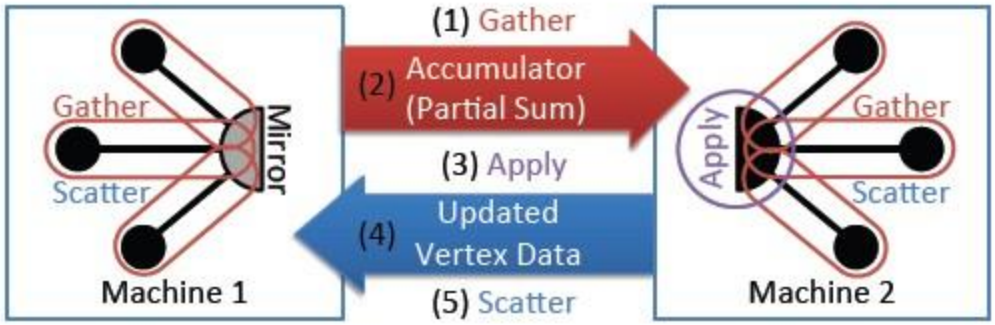
```
// scatter_nbrs: OUT_NBRS
scatter(Du.D(u.v),Dv):
if(|Du.delta|>ε) Activate(v)
return delta
```

PowerGraph

- 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：

$$\mathbb{E}\left[\left(1-\frac{1}{p}\right)^{\mathbf{D}[v]}\right]=\frac{1}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(1-\frac{1}{p}\right)^{d}d^{-\alpha}. \tag{5.12}$$

- 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，具体如下图所示。

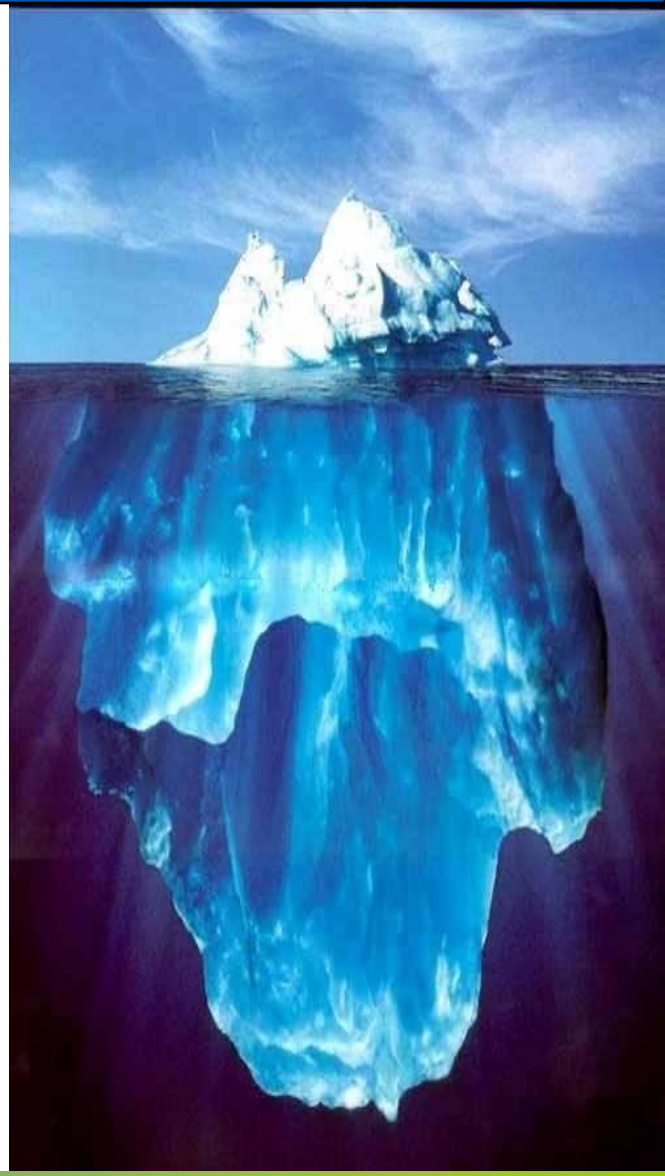


PowerGraph

- 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，可串行化异步）。
  - 1. 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edge以及vertex的修改；
  - 2. 全局异步类似GraphLab，所有Apply阶段和Scatter阶段对edge或vertex的修改立即更新到图中；
  - 3. 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。

课程提要

- 图计算简介
- 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
- 参考资料



资料

- 网上资料
  - Pregel——大规模图处理系统 http://www.cnblogs.com/panfeng412/archive/2011/10/28/2227195.html
  - PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs(OSDI'12) http://wuyanzan60688.blog.163.com/blog/static/127776163201312863021180/
  - PageRank算法在Pregel和MapReduce两种计算模型中的思路 http://wuyanzan60688.blog.163.com/blog/static/127776163201211104352543 5/
  - 被冷落的大数据热点：图谱分析 http://www.ctocio.com/ccnews/12340.html

⑩ 论文资料

--Pregel: A System for Large-Scale Graph Processing

主讲教师和助教

主讲教师：林子雨

单位：厦门大学计算机科学系
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn

助教：赖明星

单位：厦门大学计算机科学系数据库实验室2011级硕士研究生（导师：林子雨）
E-mail: mingxinglai@gmail.com
个人主页：http://mingxinglai.com

欢迎访问《大数据技术基础》2013班级网站：http://dblab.xmu.edu.cn/node/423
本讲义PPT存在配套教材《大数据技术基础》，请到上面网站下载。

Department of Computer Science, Xiamen University, September, 2013