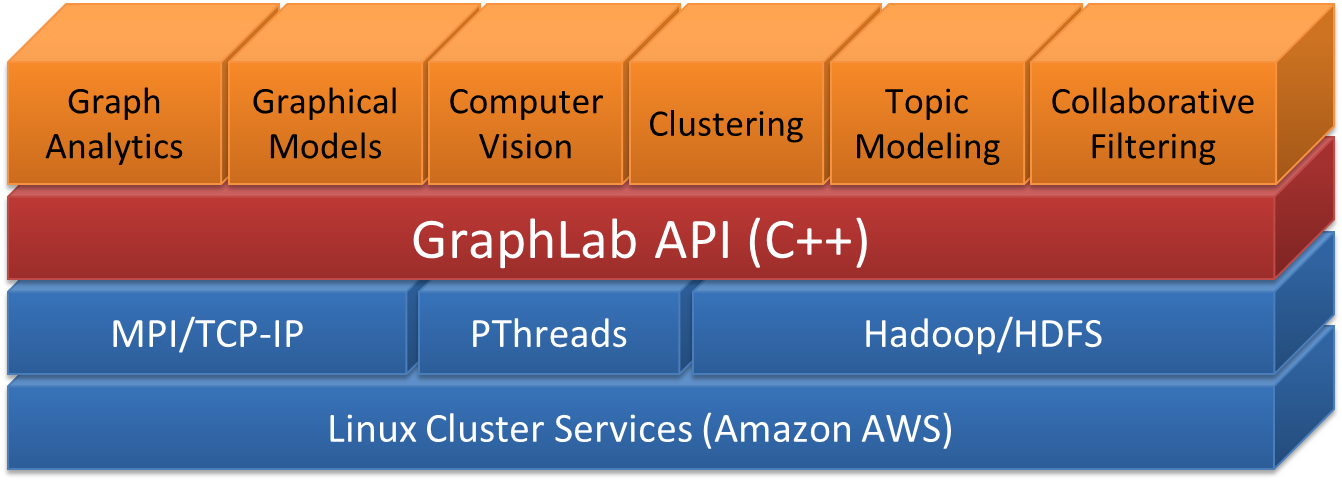
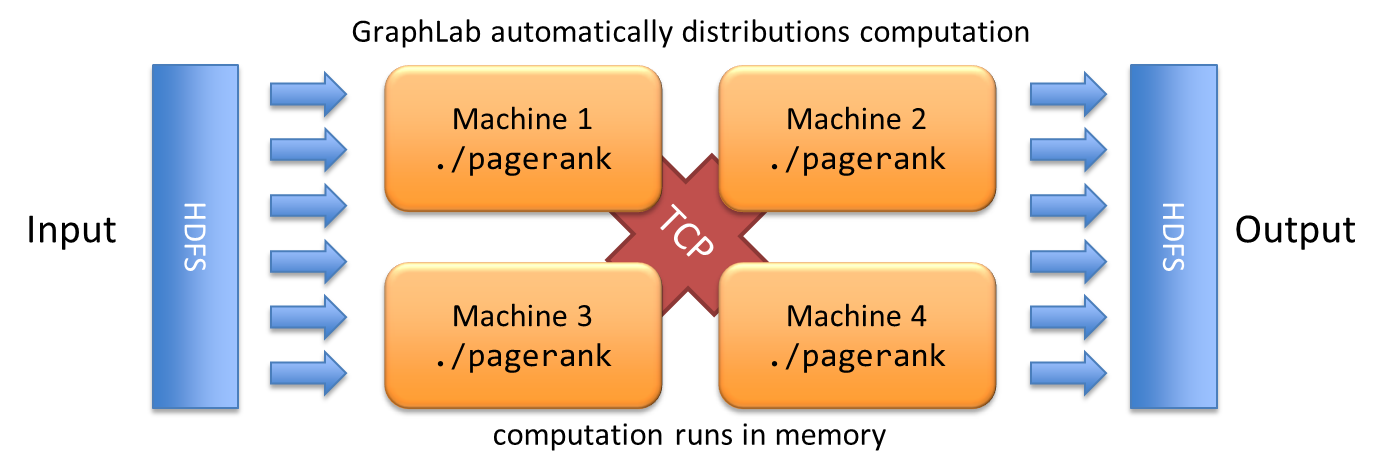
**graphlab-架构分析**

Graphlab是一个高级的图并行抽象，它高效直观的表现了计算的依赖性。Mapreduce的计算采用独立的记录。Graphlab采用依赖性的存储在大的分布式data-graph中的顶点中的记录。Graphlab中的计算用**vertex-program**表示。它能够在每个顶点上并行执行，能够和邻居顶点相互影响。

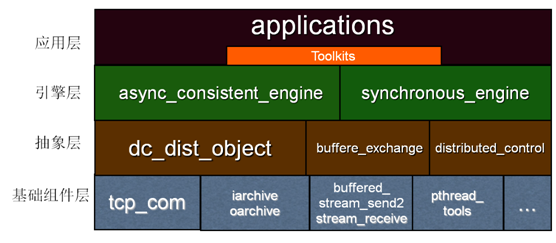
**1.       Software Stack:**





从图中我们可以看出graphlab集群中的所有机器都是等价的，无主、从节点之分。机器间采用TCP连接。

**2.       Graphlab源码结构：**



基础组件层：提供Graphlab数据传输、多线程管理等基础并行结构的组件模块，下                        面将主要介绍其通信、数据序列化、数据交换、多线程管理四个功能模块。

Graphlab基于TCP协议的长连接在机器之间进行数据通信。在Graphla初始化阶段，所有机器建立连接，将socket数据存储在std::vector<socket\_info> sock 结构中。

Graphlab使用单独的线程来接收和发送数据，其中接收或发送都可以配多个线程，默认每个线程中负责与64台机器进行通信。在接收连接中，tcp\_comm基于libevent采用epoll的方式获取连接到达的通知，效率高。

抽象层：1)      dc\_dist\_object是GraphLab对所有分布式对象的一个抽象，其目标是将分布式处理的数据对象对用户抽象成普通对象。

        2)      buffer\_exchange是基于dc\_dist\_object对需要在顶点间交换的数据提供一个容器。

        3)      distribute\_controller是基于dc\_dist\_object实现的一个整个分布式系统的控制器，提供了机器数据、顶点关系等全局信息。

        抽象层是仿照MPI完成的RPC服务。提供异步的，多线程RPC服务。实现了机器之间的通信。

引擎层：async\_consisten\_engine是异步引擎，异步的执行vertex\_programs，能够保持互斥。保证相邻的节点不会同时执行。

       Synchronous\_engine是同步引擎，同步的执行活跃的vertex\_programs。

应用层：在graphlab上的应用程序。Toolkits是graphlab上实现的应用程序。

*开发者主要构造图和实现相应的vertex\_programs。*

**3.       MPI和graphlab的关系：**

Graphlab运行在MPI并行环境上。

MPI作为引导程序启动graphlab二进制文件。

MPI初始化获取分布式并行环境中的系统参数（IP、端口号等）。

Graphlab中的rpc利用MPI初始化得到的信息进行初始化。每台机器有一个ID（0、1、2……）。

**流程：**

**1.       准备阶段：**

**a)         数据模型：graph**

**顶点：**最小的并行粒度和通信粒度。

**边：**机器学习算法中数据依赖性的表现方式。

**b)        构造：**

**数据源：**共享式文件系统（NFS）或者HDFS上的文件。

**文件内容组织形式：**以协同过滤为例：文件每一行有3列（userid、itemid、value）userid作为源顶点，itemid作为目的顶点，value为边的权重表示边的两个顶点之间的关系。

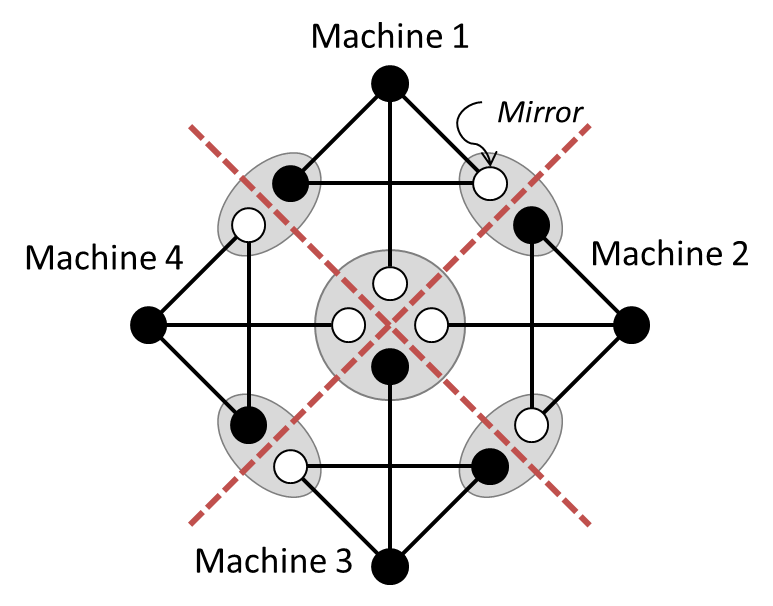
**图分割策略：**

（1）       边分割（边跨越两台机器，每台机器上各有一个边的顶点，效果不好暂时不予讨论）

（2）       点分割（顶点可以跨越多台机器，每台机器上只有一份拷贝。）

**c)         点分割：**

下图中的边唯一被分到某个机器上，但是顶点可能跨越多台机器。一台机器的顶点作为master顶点，其余机器上作为mirror。Master作为所有mirror的管理者，负责给mirror安排具体计算任务;mirror作为该顶点在各台机器上的代理执行者，与master数据的保持同步。



**点分割的三种策略：**

---Random

                  将边任意的分配到每台机器上

---Greedy

                  用一个shared  objective放置边

--- Oblivious-Greedy

                  用一个local  objective放置边

d)         **具体过程：**

**数据加载：**

1.       遍历input目录下所有的文件，生成filelist。

2.       将第一个文件分配给machine 0，依次类推，将所有的文件均分给所有机器。（每个文件只能被一台机器读取）

**分配边--random：**

1.  从文件中解析出边。

2.  将边进行hash操作，所要分配的机器ID=hash值%机器数。

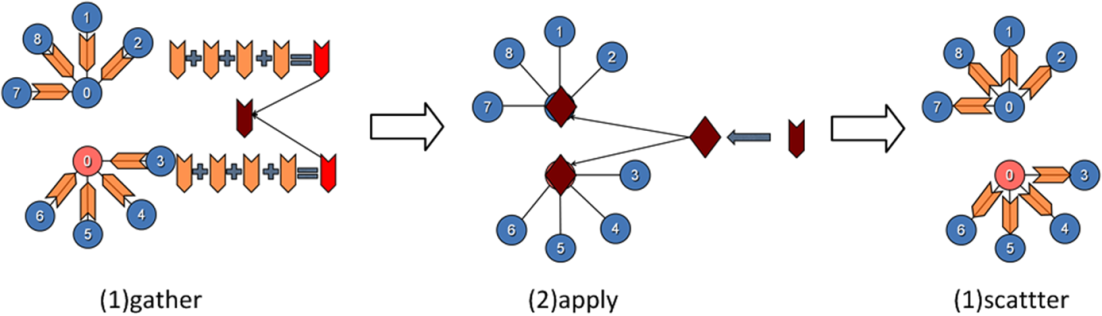
**确定master：**

将vertex进行hash操作，master所在机器ID=hash值%机器数。Master维护一个mirror列表。

**2.       计算阶段：**

**a)         执行模型：**

每个顶点每一轮迭代经过gather->apple->scatter三个阶段。

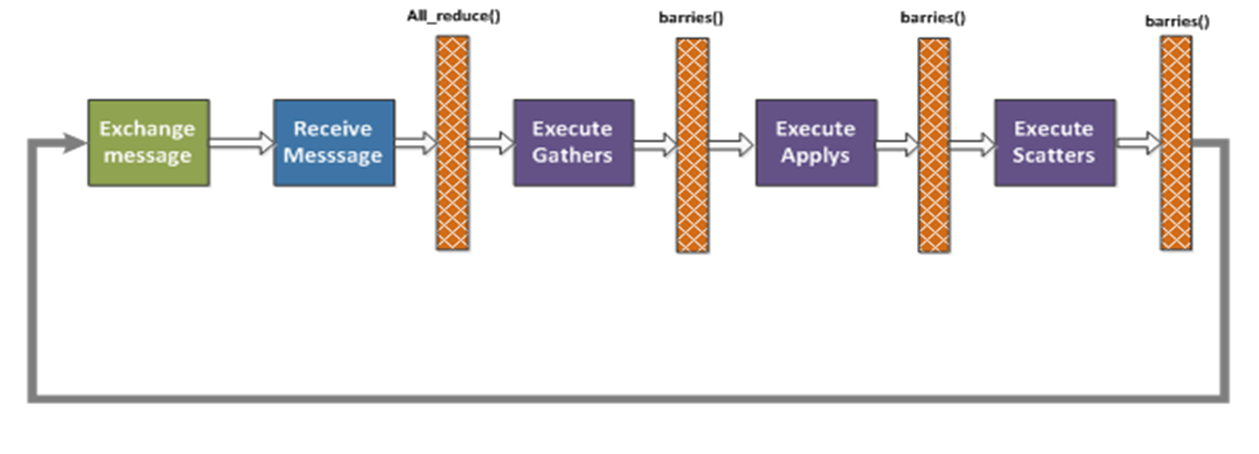


（1）       gather：从邻接顶点和自身收集数据，记为gather\_data\_i，各个边的数据graphlab会求和，记为sum\_data。

（2）       Mirror将gather计算的结果sum\_data发送给master顶点，master进行汇总为total。Master利用total和上一步的顶点数据，按照业务需求进行进一步的计算，然后更新master的顶点数据，并同步mirror。

（3）       顶点更新完成之后，更新边上的数据，并通知对其有依赖的邻结顶点更新状态。

b)         **同步引擎**



（1）       Excange message阶段，master接受来⾃自mirror的消息；

（2）       Receive Message阶段，master接收上一轮Scatter发送的消息和mirror发送的消                                           息，将有message的master激活,对于激活的顶点，master通知mirror激活，并将vectex\_program同步到mirrors；

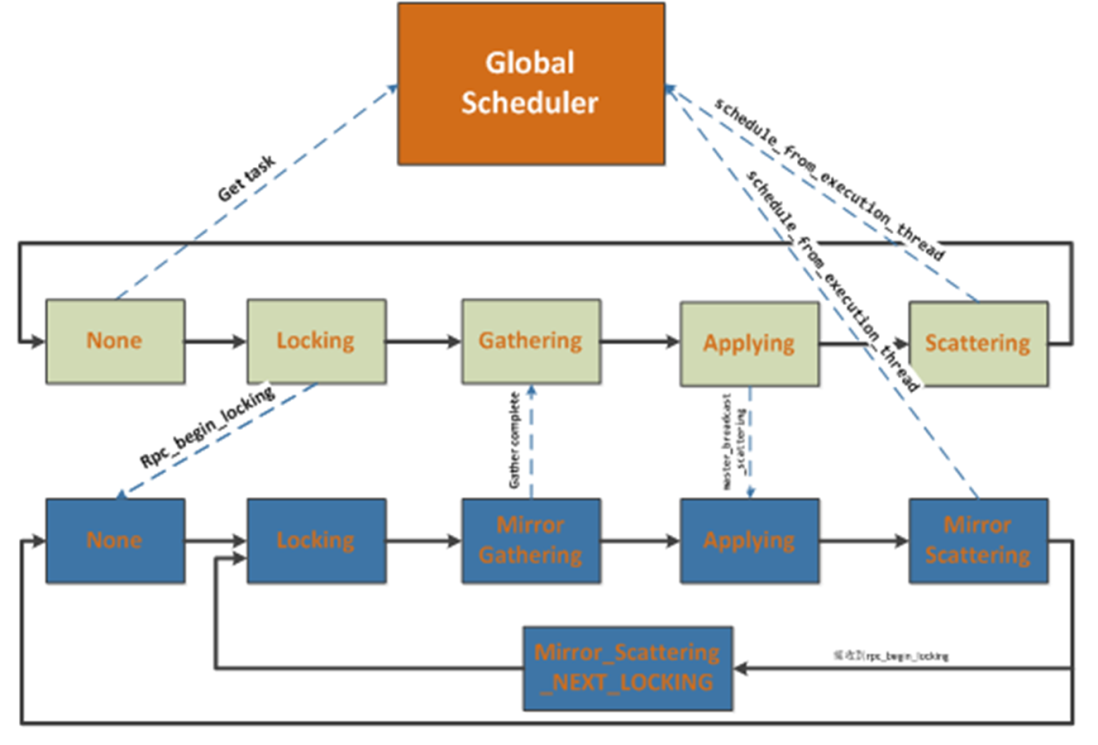
（3）       Gather阶段，多线程并行gather，谁先完成，多线程并行localgraph中的顶点，mirror将gather的结果到master；

（4）        Apply阶段，master执行apply（apply()），并将apply的结果同步到mirror (sync\_vertex\_data()).

（5）       Scatter阶段，master和mirror基于新的顶点数据，更新边上数据，并以signal的形式通知相邻顶点。

**c)         异步引擎**

异步引擎中，每个顶点是消息驱动的状态机。



（1）       在每一轮执行开始时，Master从全局的调度器(Sceduler)获取消息，获取消息后，master获得锁，并进入Locking状态。同时，master通知mirror获取锁，进入Locking状态。

（2）       master和mirror分别进行Gathering操作，mirror将gathering结果汇报给master，由master完成汇总。

（3）       master完成applying之后，将结果同步到mirror上。

（4）        master和mirror独立的执行scattering，执行完成之后释放锁进入None状态，等待新的任务到来。

（5）        mirror在scattering状态时，可能再次接收到来自master的locking请求，这种情况下，mirror在完成scattering之后将不会释放锁，而直接进入下一轮任务中。

**3.       kmeans实例---**使用同步引擎

**数据源：**

*（1）点信息.txt：*

<点id，三维数据>

1， 13.5， 16.1， 18.7

2， 3.6，  42.6， 82.1

3， 23.1， 41.3， 19.5

……

*（2）边数据.txt（点之间的关系权重，现实中每个物品之间都会存在相关性，在对它们进行分类时，会互相影响。）：*

<点id，点id，点之间的权重>

1， 3， 10

1，    5， -2

2，    4， 4

……

**读取数据信息完成图的构造。**

**Vertex-program：**

**Gather：**收集邻居节点的信息（所在的cluster），以及所在边的权重。Master通知   mirror顶点执行gather操作。并将gather告知master，master完成汇总。

**Apply：**重新计算顶点所在的cluster。根据与其他cluster的距离（使用某种距离公式）和邻居顶点的信息（如：上一次所属的cluster）以及他们之间的关系（边的权重）计算。Master执行apply操作，并将结果告知mirror。

1.       根据距离公式：计算与cluster-i的距离d-i。

2.       考虑邻居节点：如果某邻居节点属于cluster-i，则d-I -=weight（这两个点的权重）。

3.       若d-j最小，则这个顶点属于cluster-j

**Scatter：**无计算。

重新计算cluster中心。

**完成一次迭代，若满足某种条件则结束程序，否则继续循环迭代。**