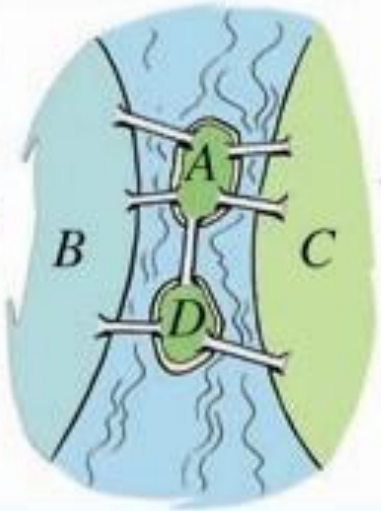
Lecture 15 图并行计算框架

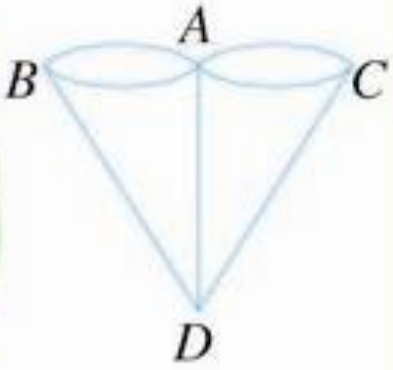
- 图计算问题
- BSP图计算模型
- 图计算架构

图计算问题

七桥问题

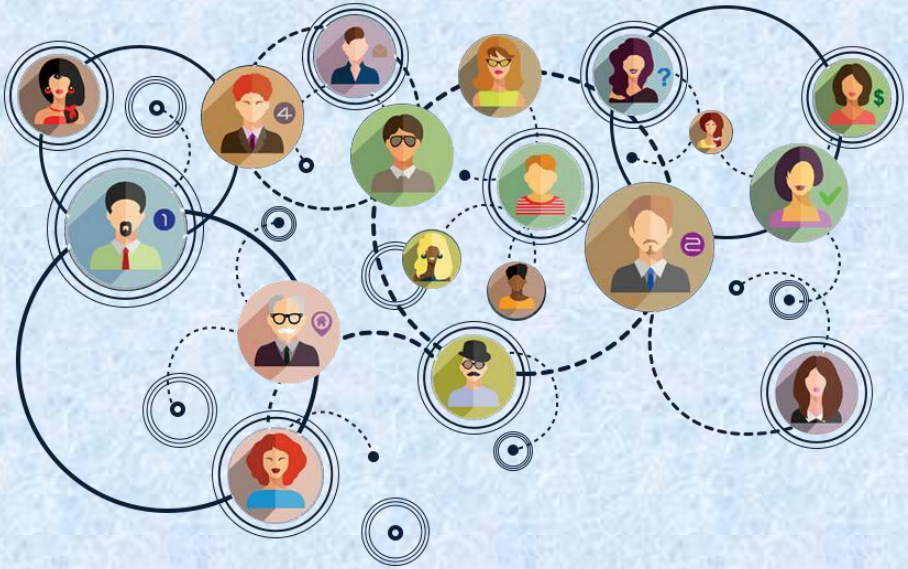
18 世纪东普鲁士的哥尼斯堡城，有一条河穿过，河上有两个小岛，有七座桥把两个岛与河岸联系起来（如下图）。有人提出一个问题：一个步行者怎样才能不重复、不遗漏地一次走完七座桥，最后回到出发点。后来大数学家欧拉把它转化成一个几何问题（如右图）——一笔画问题。

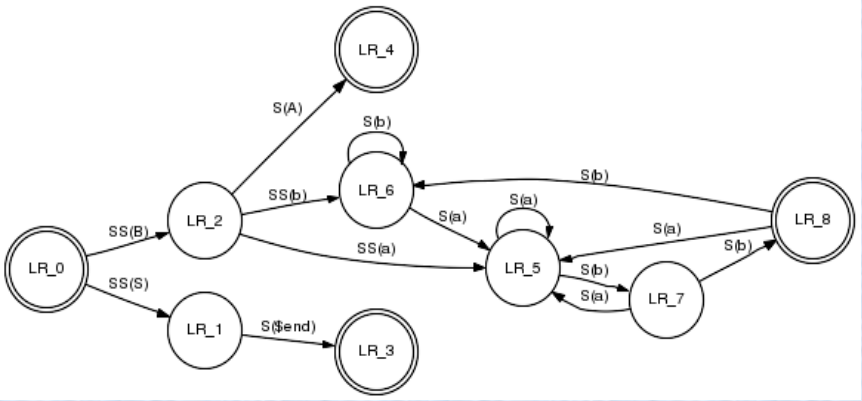




网络图计算

大型图（像社交网络和网络图等）常常作为系统计算的一部分，图计算问题包括最短路径、集群、网页排名、最小切割、连通分支等。Google报道有20%数据是采用图计算模型处理。





图计算基本概念

图（graph）：由非空顶点（vertex）集合V和边（edge）集合E组成的二元组（V, E）称为图，记为G=(V，E)。

- ✓ 无向图(undirected graph)：E中的元素称为无向边或简称边(edge)。
- ✓ 有向图(directed graph)：E中的元素称为有向边(directed edge)，也简称边或弧 (arc)。
- ✓ 简单图(simple graph)：任意两顶点间最多只有一条边，且不存在自环的无向图称为简单图。

顶点度(degree)：图G=(V, E)的顶点v的度是与v相连的边的数目（自环边计两次），记为d(v)。

图计算基本问题

子图相关问题：包括子图同构问题，哈密顿回路问题，最大团问题，最大独立集问题，平面图判定，重构猜想等

染色问题：包括点色数，边色数，色多项式，四色问题，完美图问题，列表染色问题等

路径问题：如柯尼斯堡七桥问题，哈密顿回路问题，最小生成树问题，中国邮路问题，最短路问题，斯坦纳树，旅行商问题等

网络流问题：包括最大流问题，最小割问题，最大流最小割定理，最小费用最大流问题，二分图及任意图上的最大匹配，带权二分图的最大权匹

覆盖问题：包含最大团，最大独立集，最小覆盖集，最小支配集

大数据相关的图计算方法

- 定义图数据格式，包括输入数据和输出结果格式
- 建立图计算模型与算法
  - 对于实际问题抽象出图计算模型
  - 图算法设计（全局的循环迭代（iteration））
  - 数学表达
- 图并行计算的实现
  - 图分割
  - 任务及计算资源调度
  - 计算迭代步骤
  - 同步与通讯机制

图数据结构

目前的大规模图数据应用主要采用简单图和超图两种数据模型，二者的组织存储格式略有不同。这两种模型都已处理有向图和无向图，默认情况下是有向图，而无向图中的边可以看作是两条有向边，即有向图的一种。

简单图模型的常用存储结构包括：

- 邻接矩阵；
- 邻接表；
- 十字链表；
- 邻接多重表

图计算例子

下图的有向图中寻找顶点1到顶点3的最短路径问题，可以将图表征为邻接矩阵A，再构建图向量x，而上述图最短路径问题就转化为矩阵的迭代计算问题，计算结果就是图中两点间的最短路径。

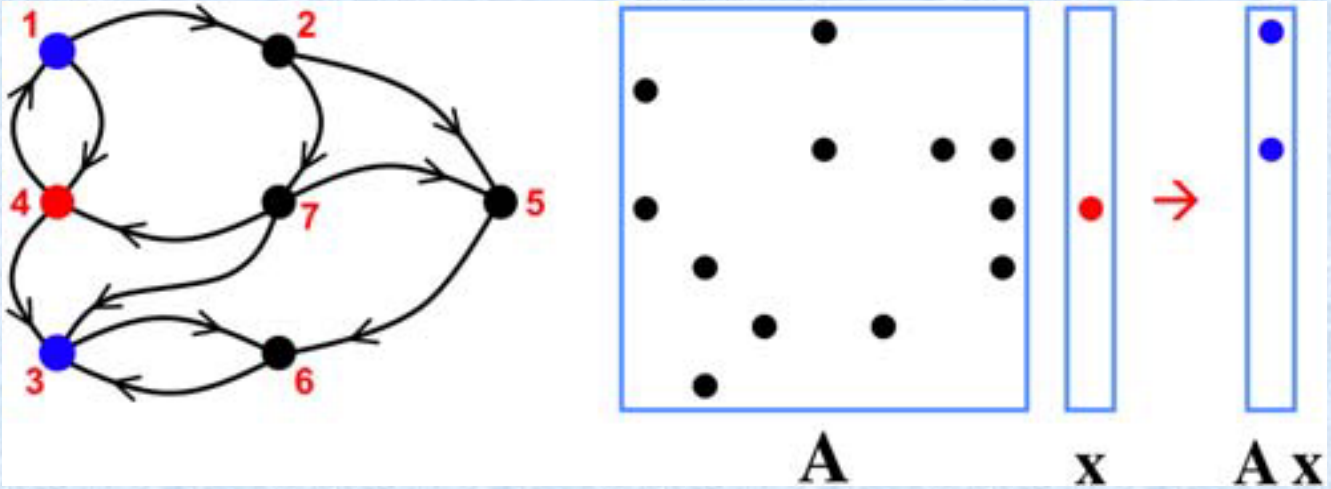
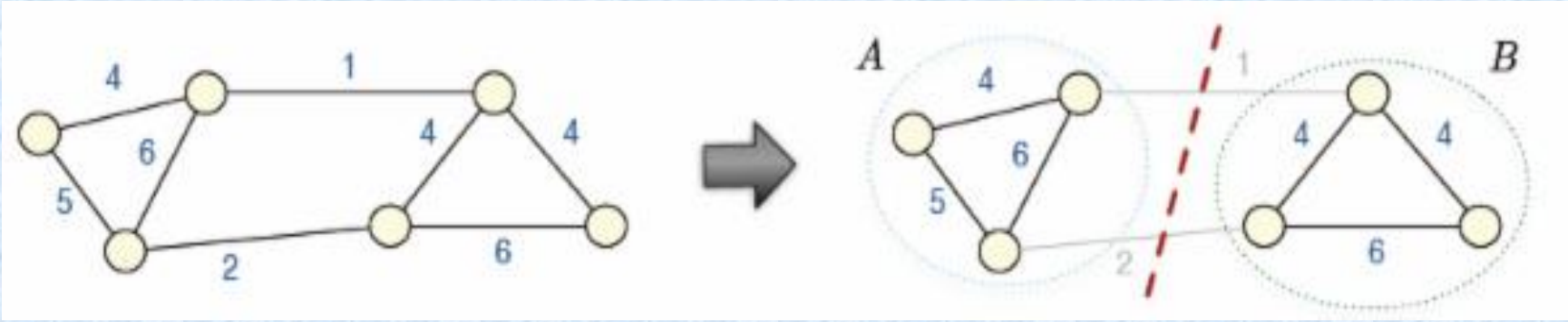


图 2-2 图问题与矩阵的转化

图分割难题

出于大数据计算目的，我们需要把左边包含6个顶点的大图分割成右边各自包含3个顶点的A和B两个小图。这种图分割（graph cut）带来的难题是：

- ✓ 被切断的边（图中编号为1和2的两条边）所代表的的特征值该如何处理？
- ✓ 在原图中相连的顶点（被切断的边连接的两端顶点）在分割成的子图中不再相连，算法设计该如何考虑？



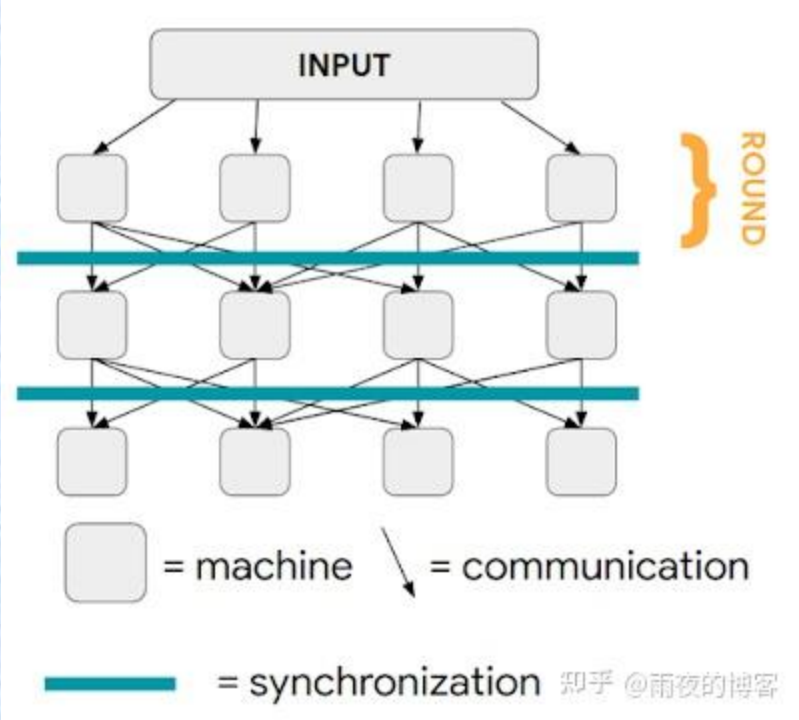
图分割方式

点切分：通过点切分之后，每条边只保存一次，并且出现在同一台机器上。邻居多的点会被分发到不同的节点上，增加了存储空间，并且有可能产生同步问题。但是，它的优点是减少了网络通信。

边切分：通过边切分之后，顶点只保存一次，切断的边会打断保存在两台机器上。在基于边的操作时，对于两个顶点分到两个不同的机器的边来说，需要进行网络传输数据。这增加了网络传输的数据量，但好处是节约了存储空间。

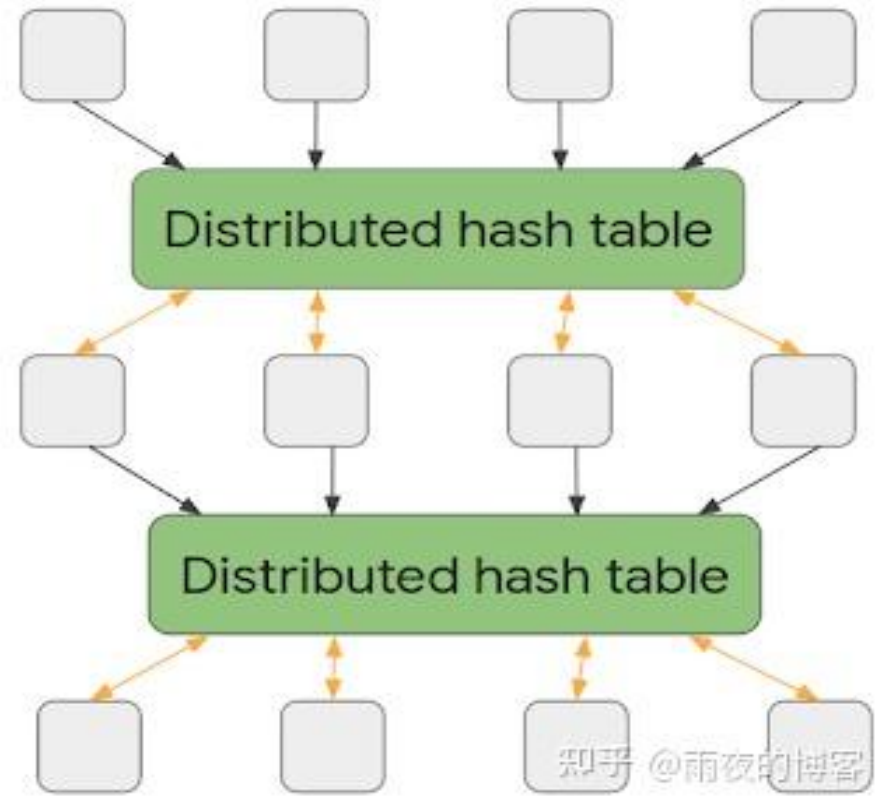
图计算MPC模型 (Massive Parallel Computing)

一组机器节点通过同步轮次中的消息传递进行通信。一轮中发送的消息在下一轮开始时传递，并构成该轮的整个输入（即，机器不保留从一轮到下一轮的信息）。在第一轮中，可以假设输入随机分布在机器上。目标是最小化计算轮数，同时确保每轮机器之间的负载平衡。

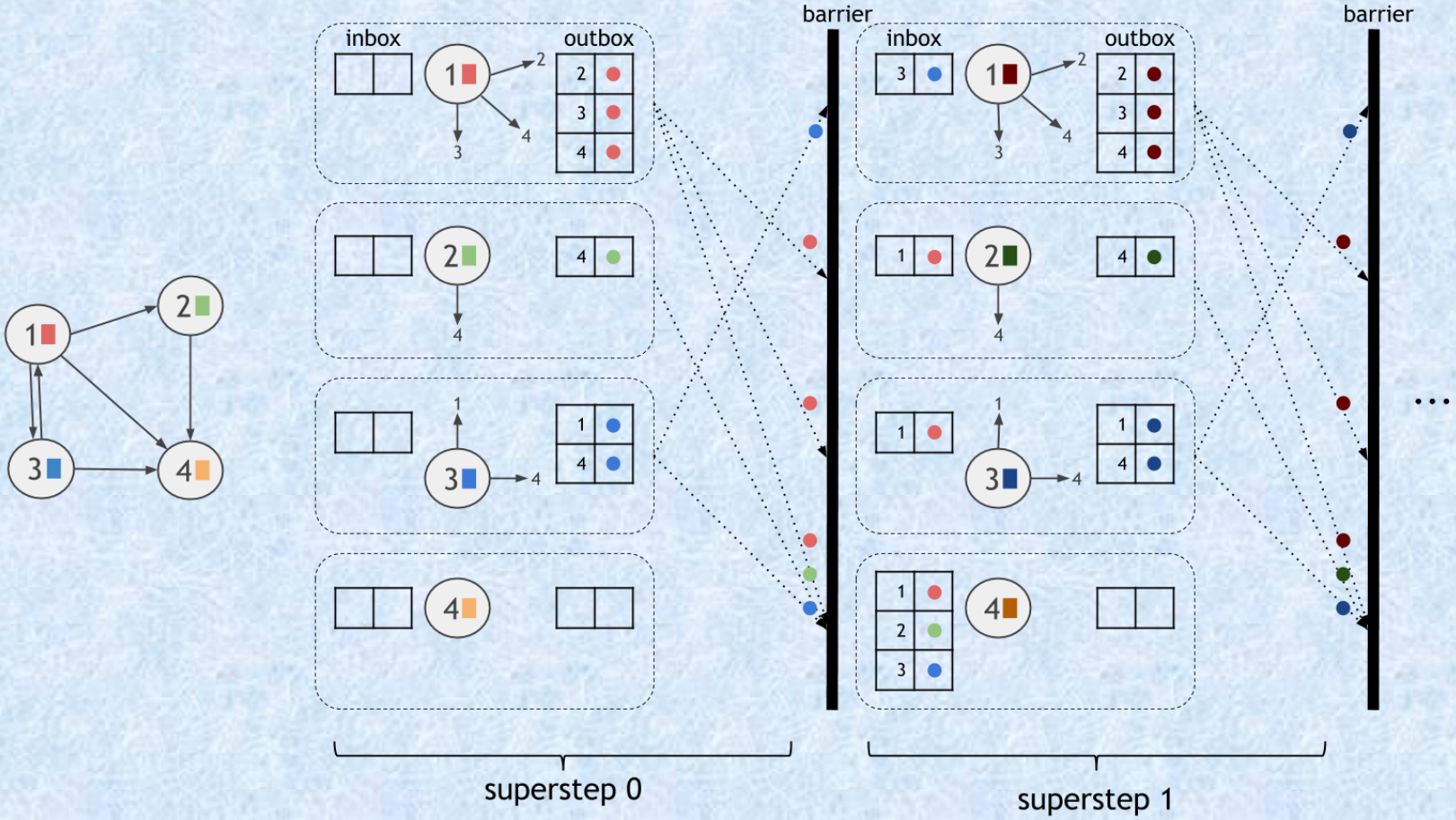


自适应大规模并行计算(AMPC)模型

机器节点每轮写入一个只写分布式哈希表，而不是通过消息进行通信。一旦新一轮开始，前一轮的哈希表变为只读，并且新的只写输出哈希表变为可用。



每一轮计算(superstep)各节点的inbox和outbox



BSP模型

BSP（Bulk Synchronous Parallel）整体同步并行模型是英国科学家Leslie G. Valiant于20世纪80年代提出的一个并行计算逻辑概念模型。其组成包含三个部分：

- 组件：每个组件由处理器和存储器组成
- 路由器：用于实现各组件之间点对点的消息传递
- 全局时钟：用于同步全部或部分的组件

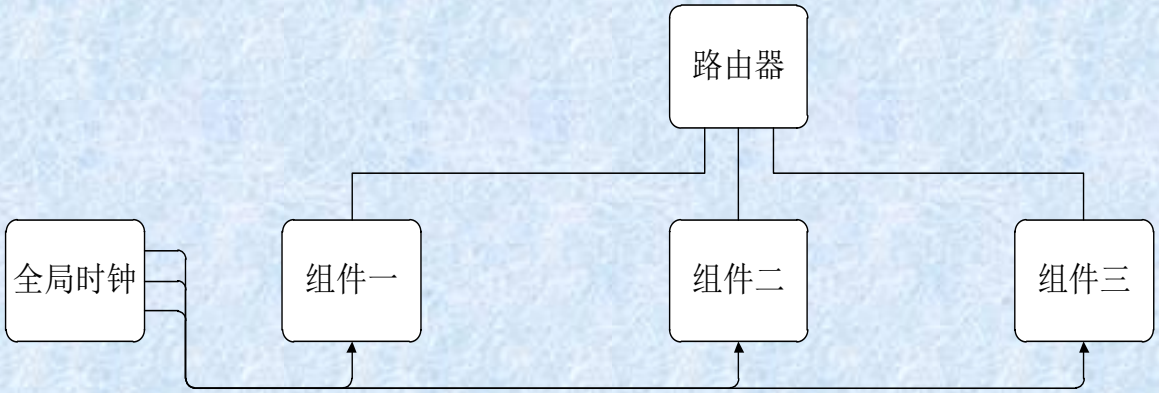


图 13-9　BSP 逻辑结构组成

BSP超步（Superstep）

BSP的核心思想是: 将任务分步完成，通过定义SuperStep (超步)来完成任务的分步计算。也就是将一个大任务分解为一定数量的超步，而在每一个超步内各计算节点(即组件，Virtual Processor代表) 独立地完成本地的计算任务，将计算结果进行本地存储和远程传递以后，在全局时钟的控制下进入下一个超步。

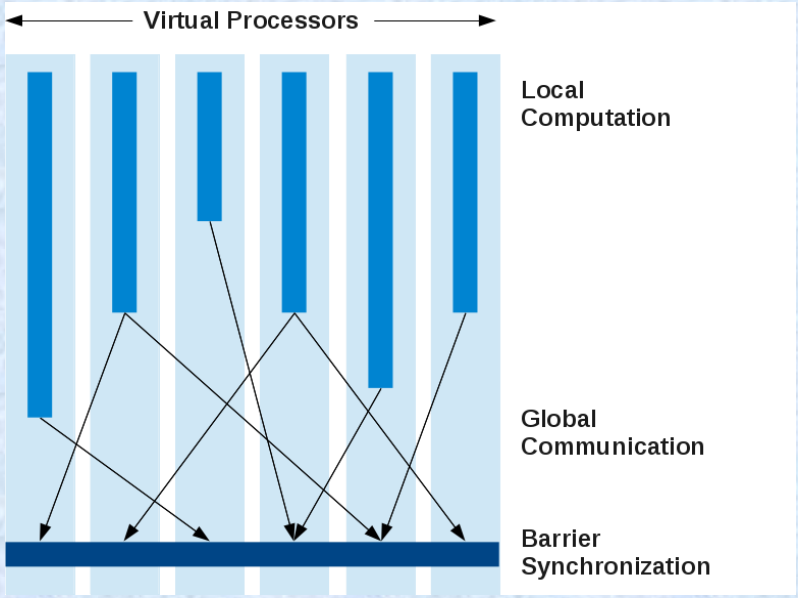


图 13-10　BSP 的超步（SuperStep）结构

BSP计算过程

本地计算：在一个超步内，处理节点（Virtual Processor）从自身存储器读取数据进行计算；

全局通信：每个处理器通过发送和接受消息，与远程节点交换数据；

栅栏同步：当一个处理器遇到栅栏（Barrier）时，会停下等到其他所有处理器完成计算；每一次Barrier同步也是前一个超步的完成和下一个超步的开始。

计算流程

- 不同超步的计算单元可以设置为相同或不相同（比如在Superstep 0，Superstep 1这两步内，Peer1与Peer 2执行不同的计算步骤A和D，但Peer 5与Peer 6却一直执行相同的计算步骤C）；
- 某些进程在特定的超步中可以不必进行障碍同步（比如在Superstep 0，Superstep 1之间，BSP Peer1与BSP Peer 2在遇到Barrier Synchronizationer 1时需进行障碍同步，但BSP Peer 5与BSP Peer 6却勿需同步。在Barrier Synchronizationer 2 时所有进程需要进行障碍同步）。

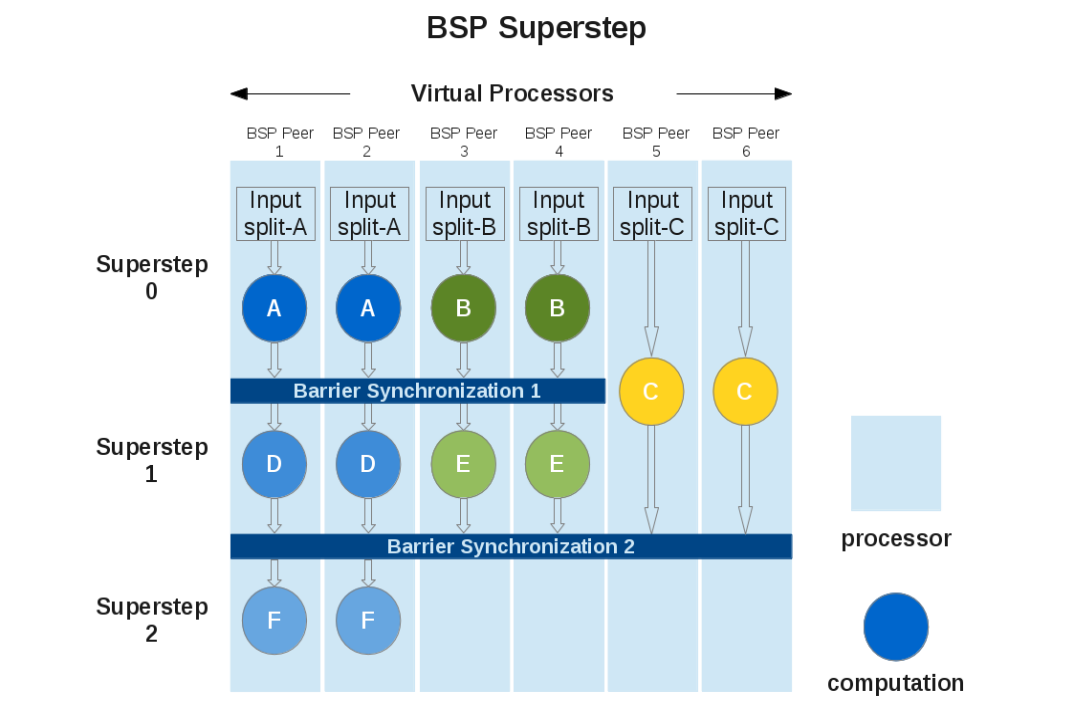
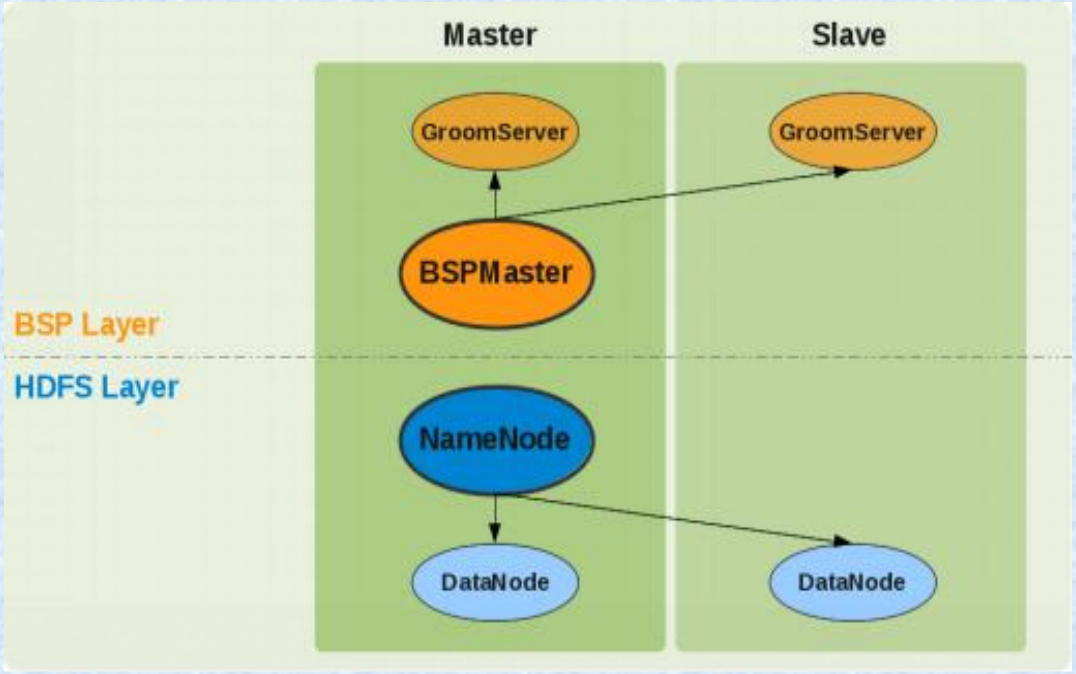


图 13-11 BSP 超步（SuperStep）实现方式

BSP计算架构

BSP并行计算架构也采用了Master/Slave模式，即在一个主节点（Master）上运行BSPMaster主程序，而在多个从节点（Slave）上运行多个GroomServer进程承担计算处理任务。

这与Hadoop 平台的Master/Slave架构（一个HDFS集群是由一个NameNode和一定数目的DataNode组成）非常相似，所以BSP模型可方便地在Hadoop/HDFS架构上实现。



Pregel图并行计算框架

Pregel是Google公司的大规模图并行计算系统，将BSP概念结构真正的用于了商业实际应用。

Pregel的计算系统仍然部署在Google计算集群上，采用Master/Slave结构，主控服务器（Master）负责计算任务的分配、调度和管理，具体负责把一个计算作业的大图分割成子图（sub-graph），然后把每个子图作为一个计算任务分发给一个工作服务器（Worker）去执行（一个Worker可能会收到多个计算任务），多个工作服务器按照超步模式完成并行计算。

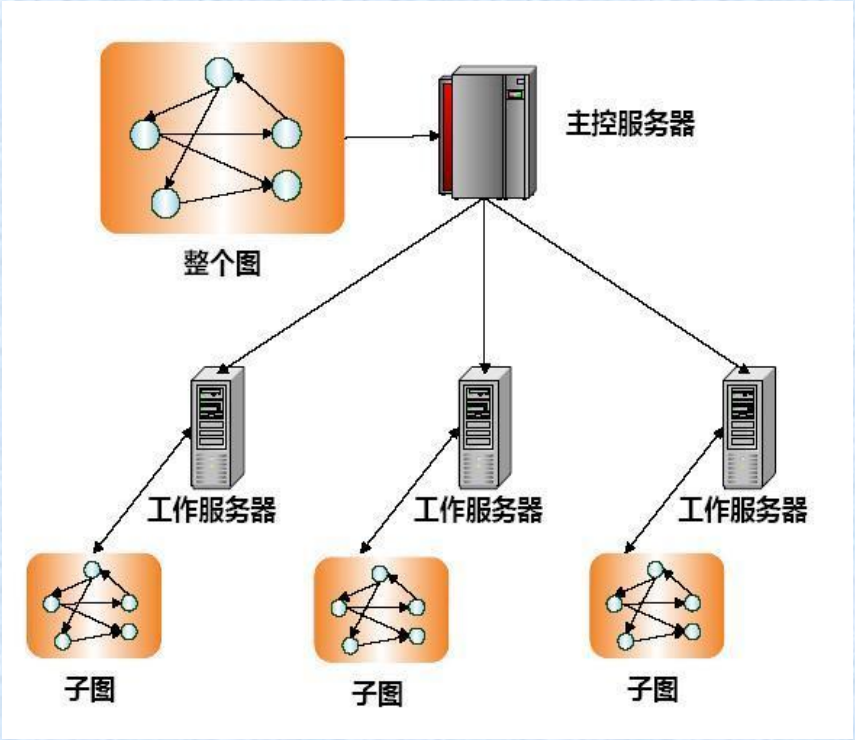


图 133-13　Pregel 的系统架构

Pregel计算架构

Pregel的计算架构如图所示，用逻辑上的Master节点来实现全局时钟的同步及超步的分界，用逻辑上的Worker节点来实现BSP模型的组件的计算功能。Pregel的计算架构包含如下要素：

Mater：
- 图分割及用户输入数据
- 任务分配调度
- 容错机制

Worker：
- 执行计算任务
- 节点间通信

持久化数据：
- 写入分布式文件系统（GFS）

中间数据：
- 存在Worker本地磁盘上

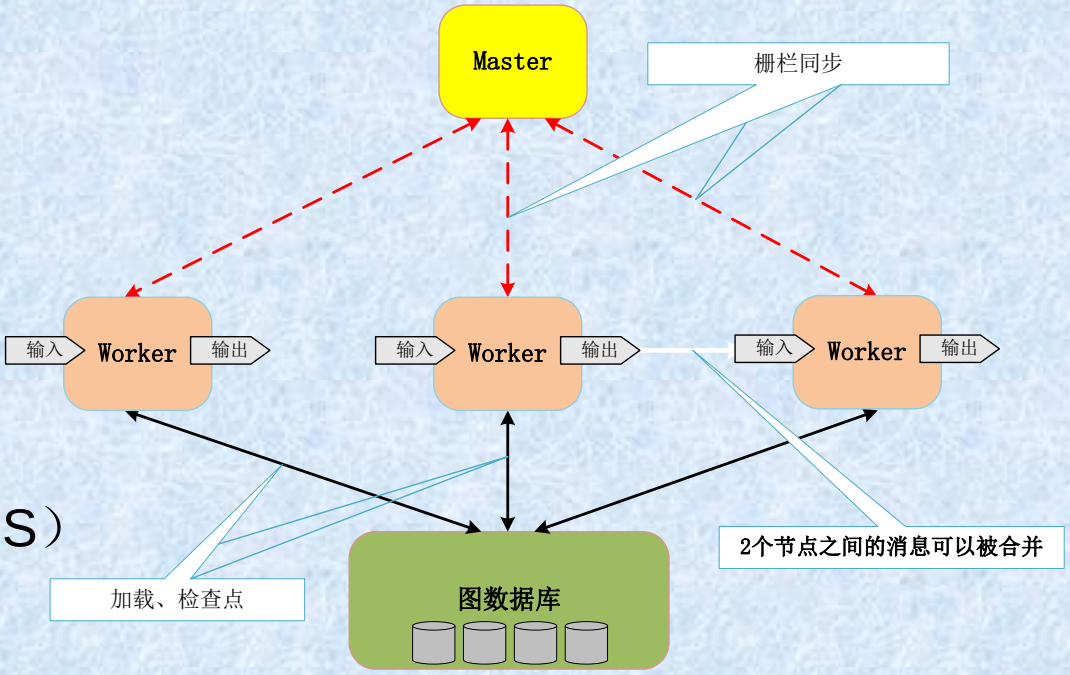


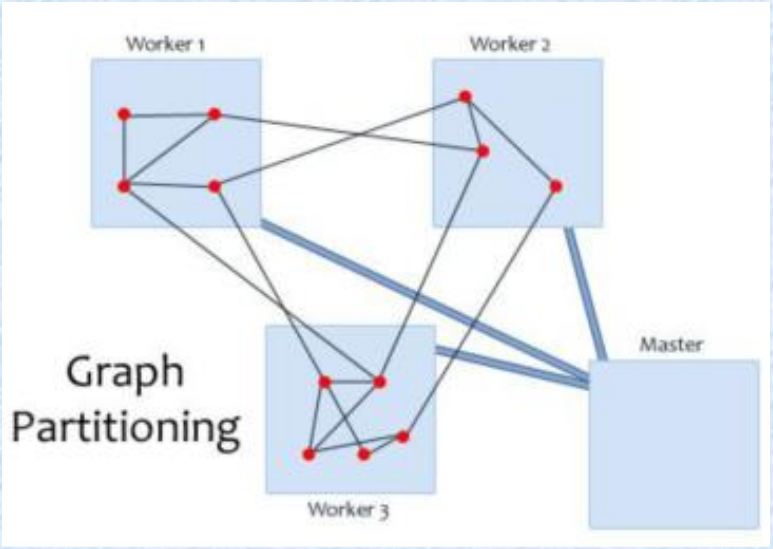
图 13-14　Pregel 的计算架构

Pregel图划分

Master首先执行的是图划分（graph partition），即将一个大图按照某种算法（partition algorithm）划分成多个分区（partition），每个分区都包含了一部分顶点（vertex）以及以其为起点的边（edge），Master则将一个或多个分区分发给每个Worker。一个顶点被分配到哪个分区由分割算法（partition algorithm）来决定的，Pregel使用的默认分割函数为哈希函数，即:

顶点对应分区号 = hash(ID) mod N

其中，N为分区总数，ID是这个顶点的标识符。另外，Pregel也容许用户自己定义partition函数。



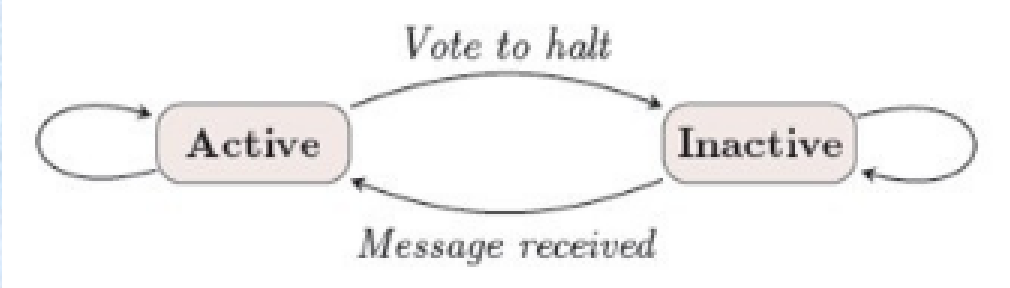
Pregel计算流程

Pregel按如下的超步（superstep）方式完成图并行计算处理：

所有节点（Worker，处理的图分区数据中包含多个图顶点vertex）对其包含的每个顶点（vertex）的计算、状态更新、顶点间同步通信都是基于超步superstep来组织；

在一个超步内，每个顶点（vertex）会调用用户定义的函数进行计算，这个计算过程是在各个顶点以并行模式进行；

所有的顶点的初始状态（superstep 0）均为“active”。一个顶点在一个超步内完成了它的计算任务，没有下一步计算要执行，就可以自己标志为“inactive”，这样它的计算函数不会再被调用，除非它又被激活；一个顶点的“inactive”状态可以为另一个顶点发送过来的消息而变为“active”（即被其他顶点的消息所激活）。



Pregel 顶点计算

每一个超步内，各顶点的计算都在节点本地进行，各顶点计算是独立的，没有对其他顶点计算结果或计算逻辑上的依赖性；

没有任何节点之外的资源竞争，因此避免了分布式异步计算系统中容易发生的deadlock；

顶点间的通信被局限在步骤之间的barrier期间完成，其含义是，每个顶点可以在超步内送出给其它顶点的消息，但这些消息不会马上处理。当这个超步结束时下一个超步开始前，所有的顶点统一处理它们各自收到的消息；

当所有的顶点都进入“inactive”状态，且没有消息传递时，Master即可决定这个作业已结束。

Pregel的顶点间通信采用了纯消息传递（message passing）模式，不包含远程数据读取或共享内存的方式，这是因为两个原因：一是消息传递模型足够满足各类图算法的通信需要；二是出于性能的考虑。在分布式环境中从远程机器上读取一个值伴随有很高的时间延迟。

实例：求向量最大值

给定一个有向连通图，图中每个顶点都包含一个值，它需要将最大值传播到每个顶点。在每个步骤中，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。当某个步骤已经没有顶点更新其包含值，那么计算就告结束。

计算要求： ① 超大规模图计算问题
② 并行计算模式
③ 避免全图遍历, one-hop通讯

系统架构： 多节点计算集群（主从架构），Master / Worker

计算模型： 基于BSP模型的图并行计算框架

计算架构： superstep / barrier communication

BSPMaster (AppMaster) / GroomServer (BSP Runner)

ApplicationManager / Container / Zookeeper

ResourceManager / NodeManager

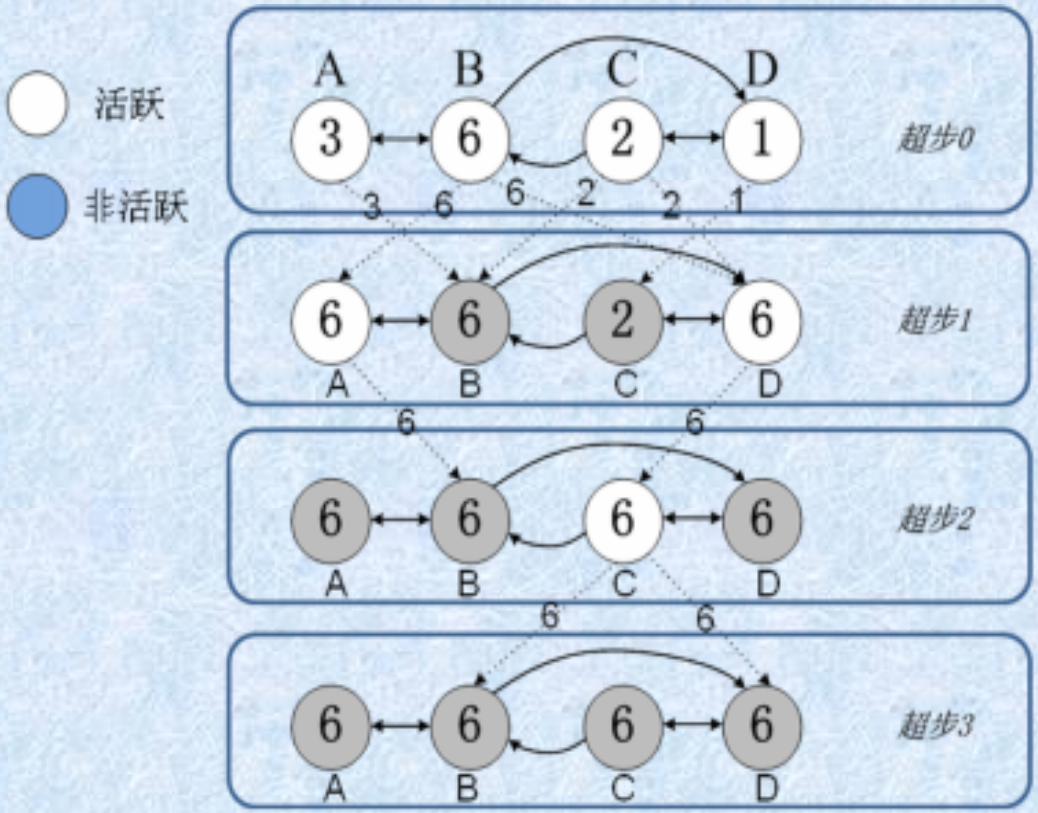
实例：求向量最大值（续）

规则：

-所有的顶点值的更新都在超步内；- 每个顶点只在超步结束时向其所有邻接点发送消息（传送顶点值）；

- 当一个顶点收到的消息中含有值比它目前值大，则用收到的最大的一个值替换它目前值，状态设置为“活跃”，否则就将状态改为“非活跃”；

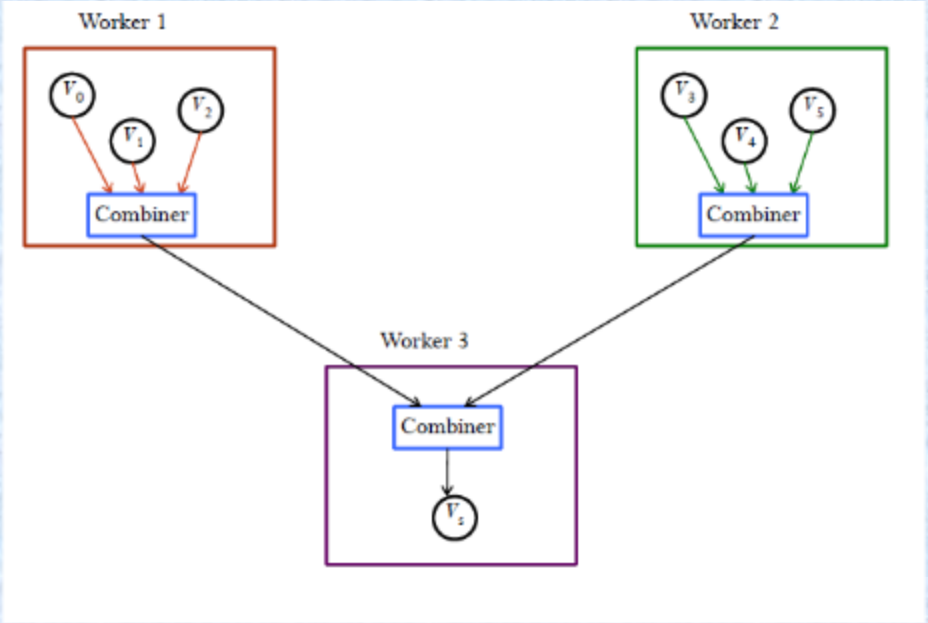
- 当所有顶点状态为“非活跃”时，计算结束。



Combiner

在超步计算之间，一个节点（Worker）上的多个顶点（vertex）可能同时向另一个节点（Worker）上的顶点发送消息，比如图，在超步0与超步1之间，顶点B和C都向顶点D发送消息。但在某些算法中，接受顶点需要的并不是每一个发送顶点的单独值，而可能是其中的最大值、或是求和值，这种情况下，Pregel提供了Combiner机制来合并发出消息，使得多个顶点发给同一目标点的多个消息合并成一条消息，从而减少消息传递开销、降低网络流量负担。

这种Combiner功能可以在发送端实现（将多条发出消息合并为一条），也可以在接收端实现（将接收到的发送给同一顶点的多条消息合并为一条），如图所示。应注意到接收端Combiner机制并未降低网络传送流量、而只是加快了接收端的处理速度。



Aggregator

另外，Pregel提供一种Aggregator机制来实现并行计算系统的全局通信、状态监控和数据查看。Pregel使用如图所示的树状结构来实现aggregator功能，即在一个超步S中，节点（Worker）上的每一个顶点（vertex）都可以向该节点的aggregator发送一个数据，系统会使用一种Reduce操作来聚合这些数据，产生的值在超步S结束时向更高一级的aggregator传送。

聚合产生值将会对所有的顶点在超步S+1中可见。Pregel提供一些预先定义的aggregators，如可以在各种整数和string类型上执行min，max，sum等操作的aggregator。

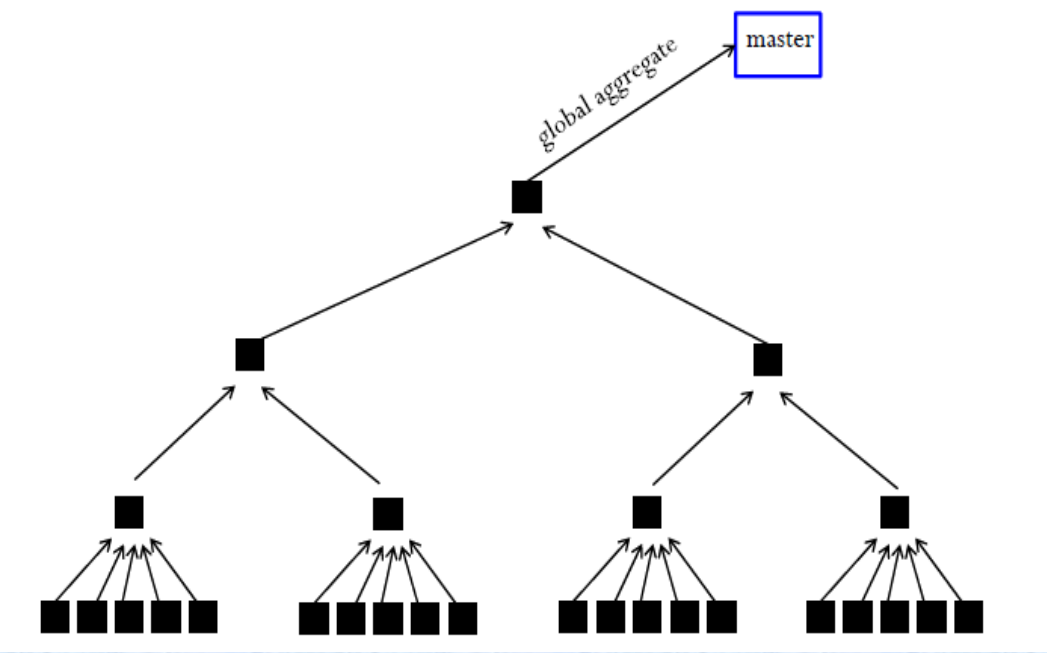


图 13-20　Aggregator 的树状结构

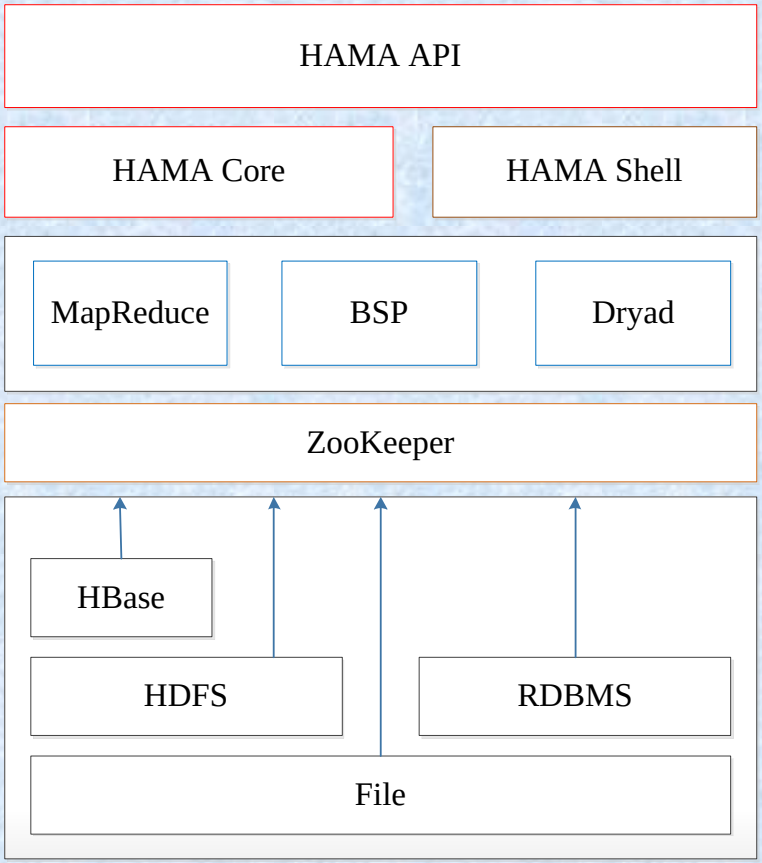
开源图并行计算框架Hama

Hama（取Hadoop Matrix的前两个字母组合）是韩国人Edward J. Yoon于2008年发起的一个基于BSP模型的图计算Apache开源培育项目[11]，并在2012年成为Apache的正式项目（http://hama.apache.org/index.html）。

Hama实际上是一个高性能集群上基于BSP并行模型和Hadoop平台构建的分布式并行计算框架（distributed computing framework），支持如下领域的大规模数据处理计算：

- 大规模矩阵运算
- 机器学习 (K-means Clustering，Decision Tree)
- 图计算（BFS, PageRank, Bipartite Matching, SSSP，最大流最小割（MF-MC）算法等）
- 网络算法（神经网络、社交网络分析、网络实时流量监测等）

Hama软件架构



Computation Engine
(Plugged In/Out)

Distributed Locking

Storage Systems

Hama介绍

Hama支持的各类算法和应用领域如图所示。

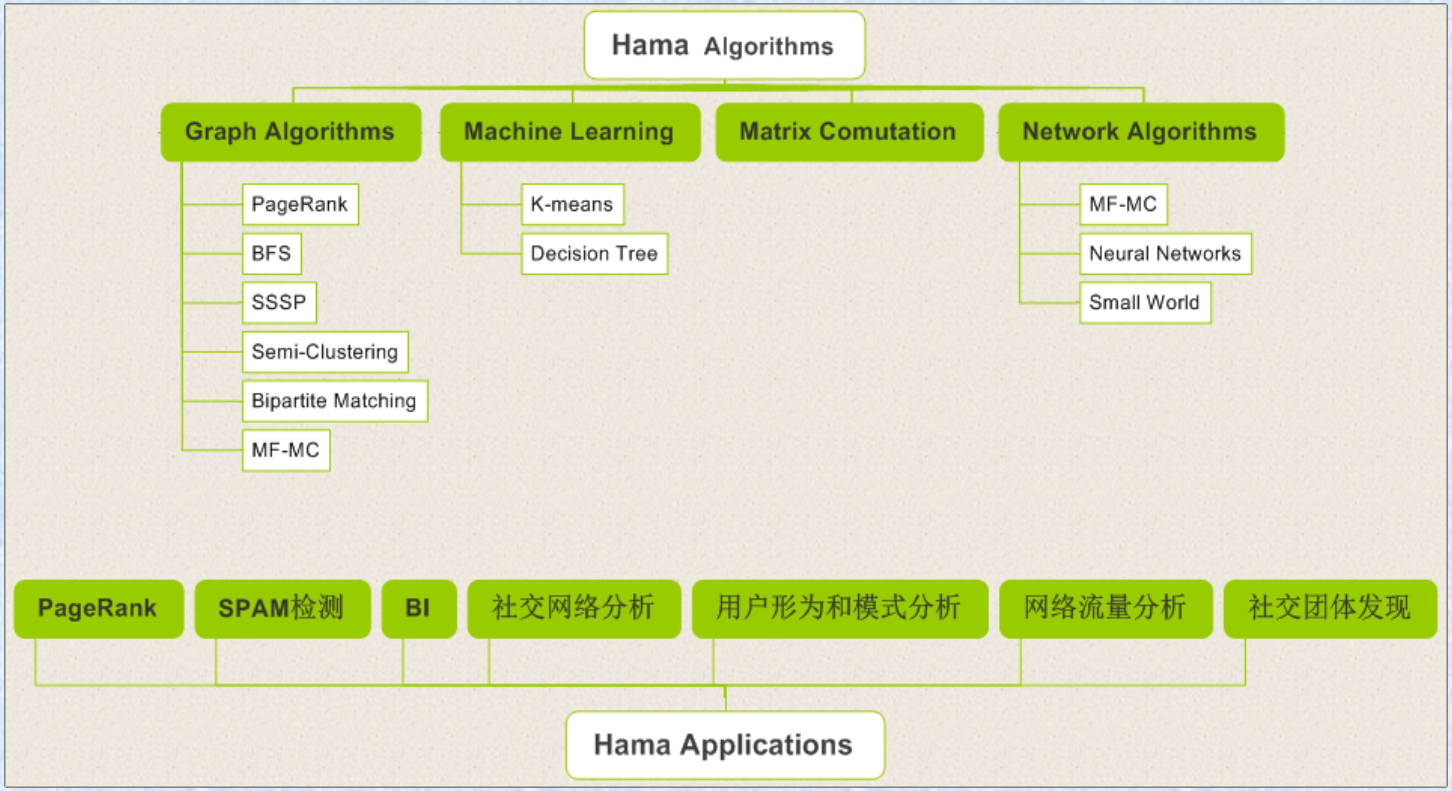


图 13-22　Hama 实现的算法和应用领域

Hama计算架构

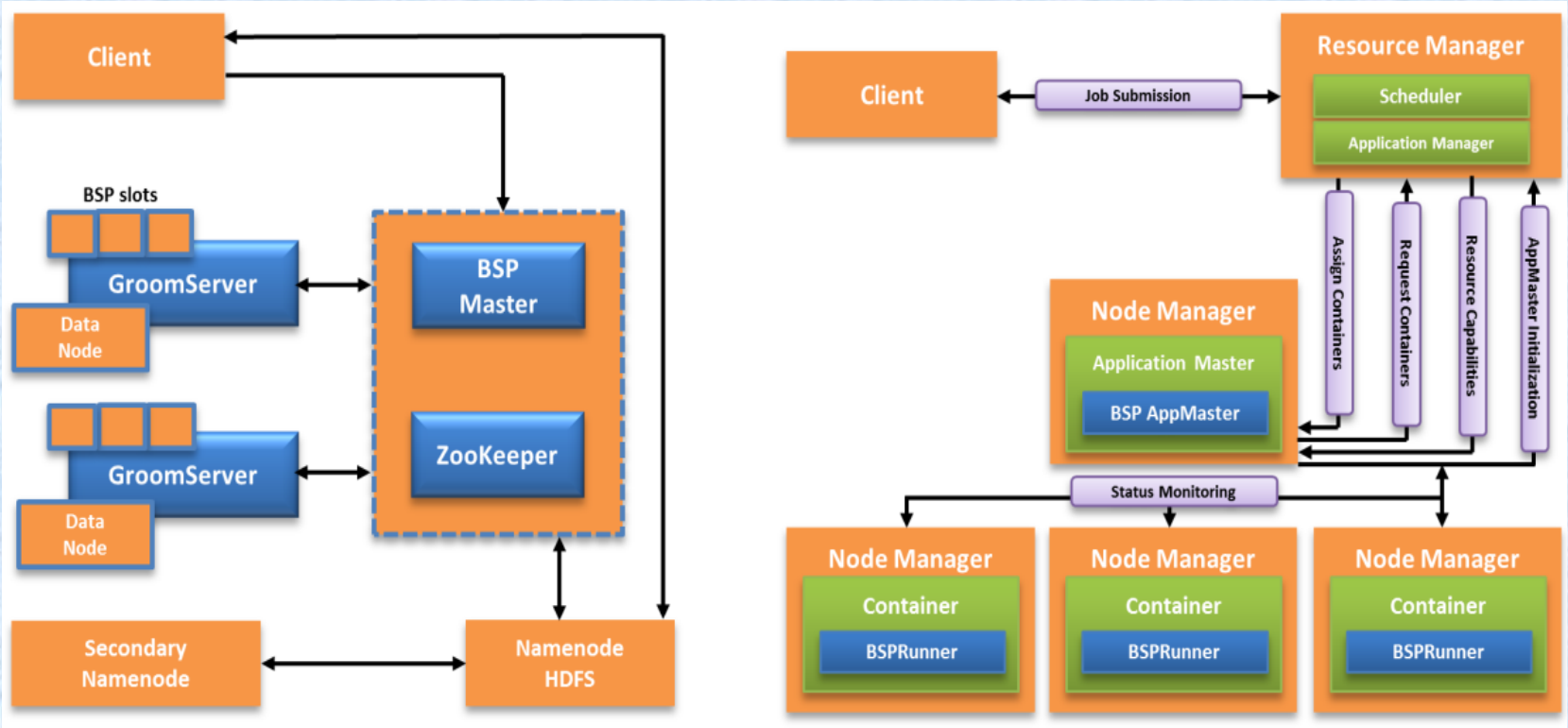
Hama的计算架构仍然采用了Master/Slave模式，即有一个主程序运行在一个集群主控节点（Master）上，有多个计算程序运行在多个计算节点（Slave）上。Hama的软件组成主要包括三部分：

- BSPMaster
- GroomServer
- Zookeeper

如图的蓝色模块所示。其中，BSPMaster（主程序）和Zookeeper（集群管理调度程序）运行于主节点（Hadoop集群的NameNode），GroomServer（计算程序）则运行在从节点上（DataNode）。

在新版本的Hama计算结构中，原来的BSPMaster被BSP AppMaster替代，GroomServer则改写成了BSPRunner，相应的程序也进行了改写，以匹配YARN运行环境。

Hama计算架构



（a）Hama Core　　（b）Hama on YARN

图 13-25　Hama 计算架构

BSPMaster

Hama主节点程序。BSPMaster主节点负责管理集群中的其他各个GroomServer从节点。在集群刚启动时，各个GroomServer节点需通过RPC在BSPMaster节点处进行注册，并向其汇报GroomServer节点当前所具有的资源数量（Task Slot的数目），BSPMaster会为每一个GroomServer及Task Slot分配ID。BSPMaster节点还负责作业的调度及分配工作，具体的计算任务则分配到GroomServer节点上运行。BSPMaster节点具体负责的工作包括：

- 维护其自身的各种状态信息
- 维护各个GroomServer服务器的状态
- 控制集群环境中的超步（Superstep）及各类计数器（Counter）
- 管理在集群中运行的作业及任务
- 调度任务到GroomServer节点，分配任务并向各GroomServer发送执行任务的指令
- 为用户提供集群的管理界面

GroomServer

GroomServer是一个运行在计算节点上的进程，负责执行计算任务（Task）和管理任务运行生命周期。每一个GroomServer都与BSPMaster进行通信，获取任务并报告状态。GroomServer需要Hadoop/HDFS运行环境支持，通常GroomServer运行在Hadoop的DataNode上，以保证获得最佳性能。GroomServer节点上，有运行具体任务的任务槽（Task Slot），与Hadoop MapReduce具有Map和Reduce两种任务不同，在Hama中只有一种任务（BSP Task），各个作业的任务最终将会在这些任务槽中来运行。每个GroomServer节点在运行过程中，会通过heartbeat方式周期地与BSPMaster节点通信，向BSPMaster节点汇报其目前空闲的任务槽数目、任务运行状态以及接收新的任务指令等。

需注意的是，GroomServer节点上还有一个重要组件BSPPeer。每个GroomServer分得的作业partition都进一步分解成基于图顶点（vertex）的计算任务（Task），每一个计算任务都有一个对应的BSPPeer来提供顶点间的通信和同步功能。

Zookeeper

Zookeeper（ZK）用来管理BSPPeer的同步，实现超步的Barrier Synchronisation机制。在ZK实现机制中，BSPPeer主要有进入Barrier和离开Barrier两种操作，所有进入Barrier的BSPPeer会在ZK提供的文件结构中创建一个临时节点（/bsp/JobID/SuperstepNO./TaskID），最后一个进入Barrier的BSPPeer同时还会创建一个ready node (/bsp/JobID/Superstep NO./ready)，然后BSPPeer进入阻塞状态等待，直到ZK上所有的node都删除后才退出Barrier。

Hama计算流程

一个Hama作业（Job）的流程首先分为三部分：JobClient的作业提交、BSPMaster的初始化与作业分发、以及GroomServer的计算任务执行，如图所示。

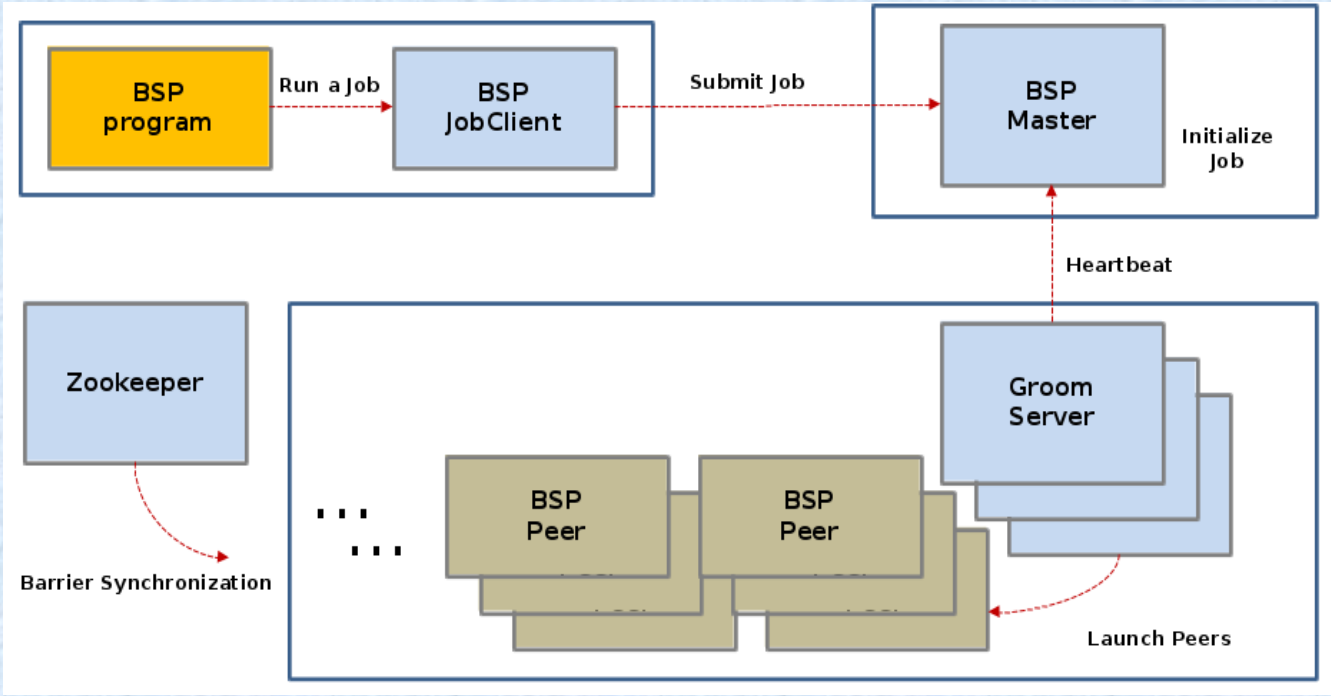


图 13-26　Hama 计算流程

Hama作业流程

右图详细描述了Hama作业的生命周期，它包含如下阶段：
作业的提交→
作业初始化→
任务分派→
任务执行→
状态更新→
作业完成

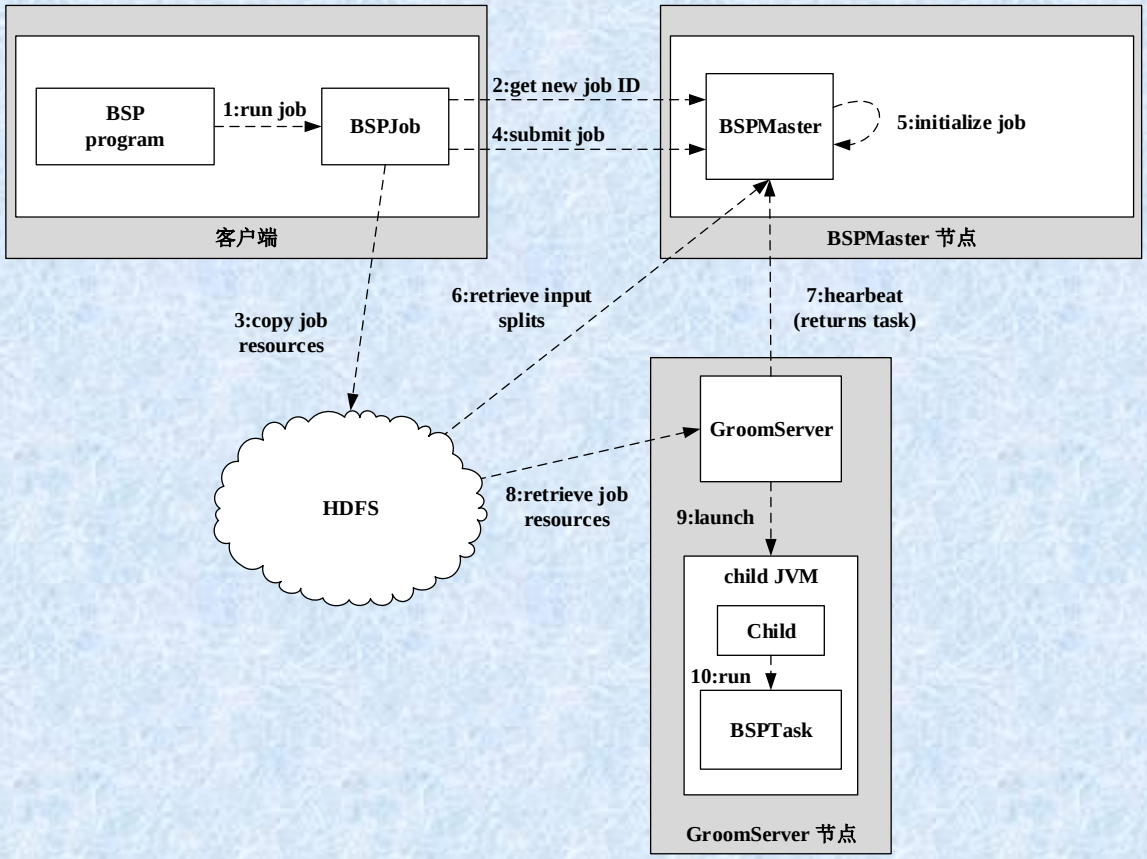


图 13-27　Hama 作业提交、分发、执行的全过程

作业的提交

在提交作业后，调用Job对象中的waitForCompletion()方法使得客户端程序等待作业完成，在这一过程中，客户端会定期的将作业的运行状态打印到控制台。作业的提交过程全部在客户端完成，遵循以下步骤：

1.向BSPMaster申请一个新的作业ID（通过RPC调用BSPMaster上的getNewJobId()方法）（图中步骤2: get new job ID）;

2.检查作业的输出设置，如果作业的输出目录未设置或已存在，作业将不会被提交，作业将会因为异常而终止;

3.为输入文件计算分片（partition），如果无法计算输入分片（如输入路径未设置），作业同样将不会被提交，作业将会因为异常而终止。计算好的作业分片信息会与其他作业运行相关的资源存放至HDFS中;

4.上述过程均成功后，与运行作业相关的文件（包括作业的JAR文件，作业的配置信息及输入文件的分片信息）将被复制到HDFS上BSPMaster的目录下，并以作业的ID作为文件名标识。作业JAR的备份度通常会很高（默认为10），以使得JAR可运行文件能够在集群中足够分散，方便GroomServer在运行作业的任务时可以快速的读取作业资源（图中步骤3: copy job resources）;

5.最后，客户端将作业提交给BSPMaster（图中步骤4: submit job）。

作业初始化

当BSPMaster接收到提交的作业后，会根据作业中的设置信息为作业进行初始化，即为作业创建一个代表运行中作业的对象JobInProgress，在JobInProgress封装了作业的任务信息及为了跟踪任务运行状况而所需的统计信息（图中步骤5: initialize job）。JobInProgress将作为作业的实际代表存在于BSPMaster节点上，并会被作业调度器“捕获”放入对应的作业等待队列中。

在JobInProgress初始化过程中，作业中的各个任务也将由JobInProgress来负责“预初始化”，首先JobInProgress会读取该作业的输入分片信息，随后JobInProgress将根据分片信息中分片的个数（Task数）为作业初始化任务，并将任务信息记录在JobInProgress的数据结构中。随后，JobInProgress会通知BSPMaster，作业已初始化完毕，等待调度。（图中步骤6: retrieve input splits）

任务调度及分派

GroomServer在运行期间会周期性的向BSPMaster发送“心跳”信息。“心跳”信息中包含GroomServer的状态，一个GroomServer可以通过“心跳”信息告知BSPMaster其当前正在运行的任务数以及剩余的空闲任务槽数目。BSPMaster会将各个GroomServer汇报上来的状态信息缓存起来，作业调度器将会使用这些信息来为作业分配具体的执行节点（图中步骤7: heartbeat (returns task)）。

在进行任务分配之前，BSPMaster必须按照作业调度算法选择作业。目前Hama只有一个先来先服务（FCFS）作业调度算法，一旦BSPMaster选定了作业，就可以为作业分派具体的执行节点了。

任务运行

经过任务的调度及分派之后，GroomServer已经被分派到了新的任务，下一步将是运行该任务。首先，GroomServer把HDFS中存储的作业JAR文件复制到本地文件系统中，同时，它也会复制运行作业所需的其他文件到本地磁盘（图中步骤8: retrieve job resources）。然后，GroomServer将读取到的JAR可运行文件解压，并创建一个TaskRunner实例来运行该任务。

TaskRunner将会在一个新创建的Java虚拟机（JVM）（图中步骤9: launch）中独立运行所分派到的任务（图中步骤10: run）。由于新建的JVM独立于GroomServer的JVM，因此在用户程序中的任何bug均不会影响到GroomServer。任务执行子线程通过通信协议与其父进程进行通信，将任务的运行情况汇报给父进程。

基于Hama并行计算框架的应用程序，均需继承自抽象基类BSP，用户算法的实现需定义在bsp方法中，该方法接受一个BSPPeer类作为参数，BSPPeer负责为任务提供输入以及输出功能，并实现各个任务之间的通信以及同步工作。BSP类除bsp()方法外，还有两个方法是可选的，即setup方法和cleanup方法，用于实现程序的准备（setup）以及清理（cleanup）工作。

作业状态更新

Hama作业通常是长时间运行的批处理作业，运行时间会从几分钟到几个小时不等。由于时间较长，因此用户能够及时的从作业的运行情况中得到反馈是至关重要的一个设计因素。在Hama的设计中，每个作业和任务都有一个状态（status）信息用来表示当前作业或任务的状态（state）（如运行，成功完成，失败等），任务执行进度以及其它的作业统计信息。

作业完成

当BSPMaster收到作业最后一个任务完成的信号后（上文提到的cleanup任务），它会把作业的状态信息设置为“成功”。然后，在客户端调用作业信息时，将“成功”标识返回给客户端。客户端在获知作业已成功运行完毕后，将在控制台上打印作业统计信息。最后，BSPMaster将会清理该作业占用的相关资源，并通知GroomServer做清理工作。

作业调度策略

Hama计算框架是通过各节点间的消息发送来达成数据的一致性的。一个作业在运行过程中发送消息的数量不仅会占用GroomServer本地内存空间，过大的消息发送量还会影响集群的网络通信性能。Hama根据作业的消息发送量把作业分为如下两类：

消息密集型作业

消息密集型作业是指在作业在运行过程中产生的消息量大于作业本身输入的数据，这类作业不仅使得内存占用率增高，而且会大量耗费集群网络带宽，很容易产生GroomServer节点过载。典型的消息密集型作业包括网页排名（PageRank）及单源最短路径（SSSP）等计算。

CPU密集型作业

CPU密集型作业也可称为非消息密集型作业，这类作业发送的消息量一般是小于其输入的数据量，在其运行过程中主要是在本地节点上进行运算，只通过少量的网络消息来达成各个计算节点所需的消息交换，甚至不需要发送网络消息。典型的CPU密集型作业如机器学习中k-meams聚类算法等。

作业（Job）管理

Hama的核心功能，主要包含作业的提交、调度及任务（Task）的分发和管理。作业调度主要由BSPMaster来完成，BSPMaster负责维护每个Job的相关信息，将一个Job的执行分解为多个Task，分配给各GroomServer。

GroomServer负责执行Task任务，并将执行状态等参数返回给BSPMaster。Task分配调度流程如图所示。

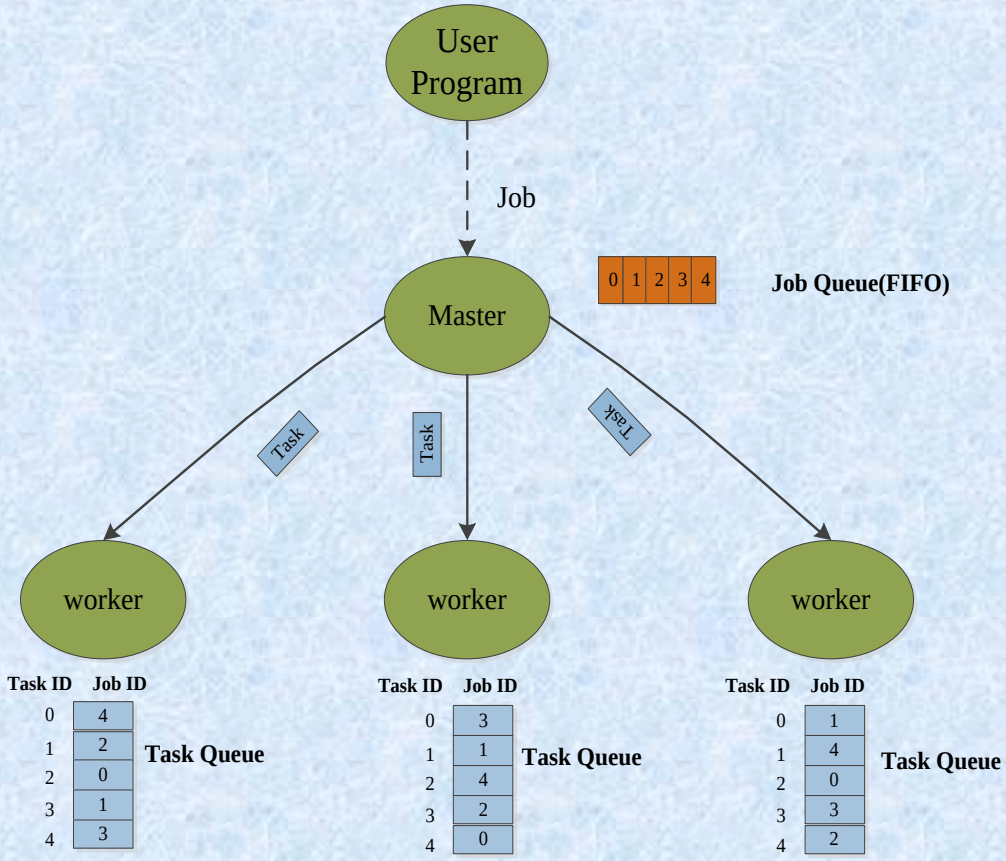
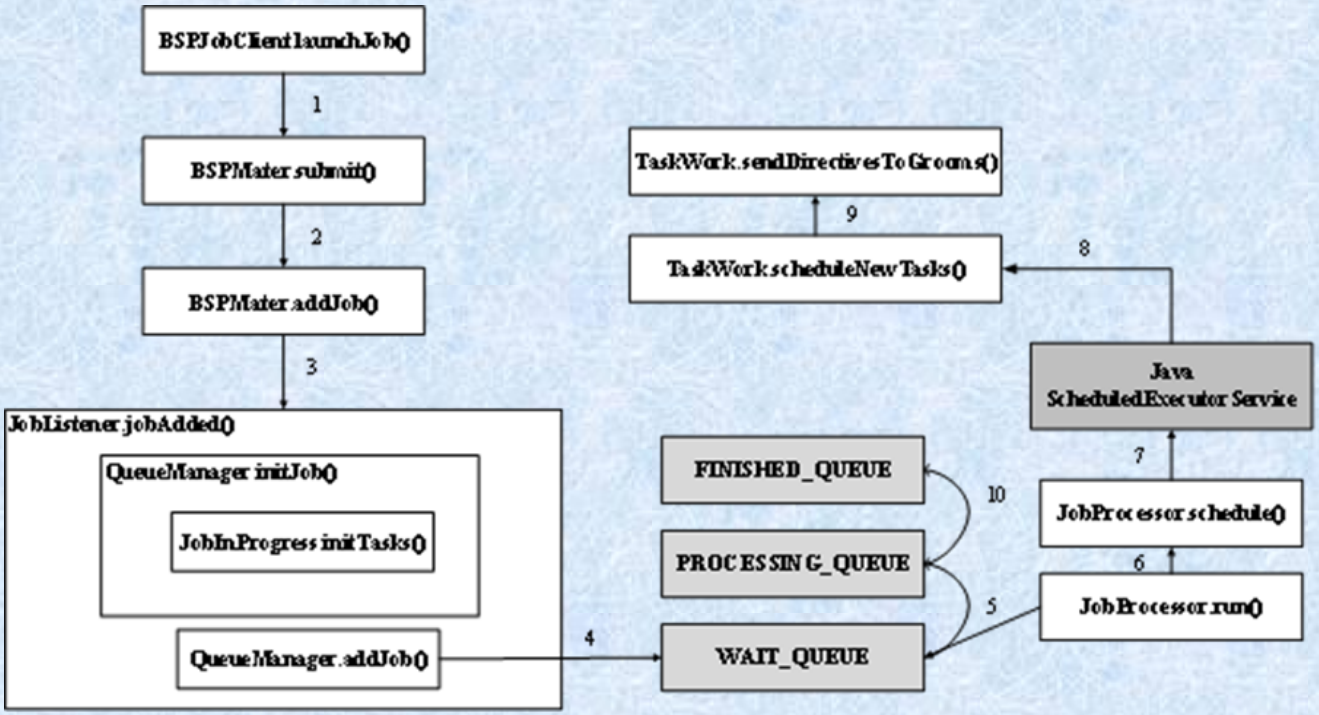


图 13-29 Hama Task 分配调度

FCFS作业调度器

使用一个FIFO队列来管理用户提交的作业，在调度作业时，会从等待队列中选取队首作业作为下一个执行的任务，并为该作业分配具体执行节点。FCFS作业调度器的工作流程如图所示（图中各方法均省略了参数名），包含如下调度步骤：



Step 1: 作业加入FIFO队列。作业首先由客户端（BSPJobClient）提交至master节点，BSPMaster会负责初始化该作业，生成一个代表其运行状态的对象JobInProgress，然后被JobListener中的jobAdded方法添加至FIFO等待队列中；

Step 2: 下一执行作业选择。作业的选择由JobProcessor完成，当FIFO队列有作业等待时，JobProcessor会从等待队列的队首摘取一个作业放入运行队列中，并调用schedule方法进行调度工作；

Step 3: 任务分配至执行节点。在任务分配过程中，TaskWork会利用注册在JobInProgress对象中的任务分配策略来为任务分配具体执行节点，然后TaskWork发出任务执行指令至具体执行的GroomServer，至此作业调度流程结束。