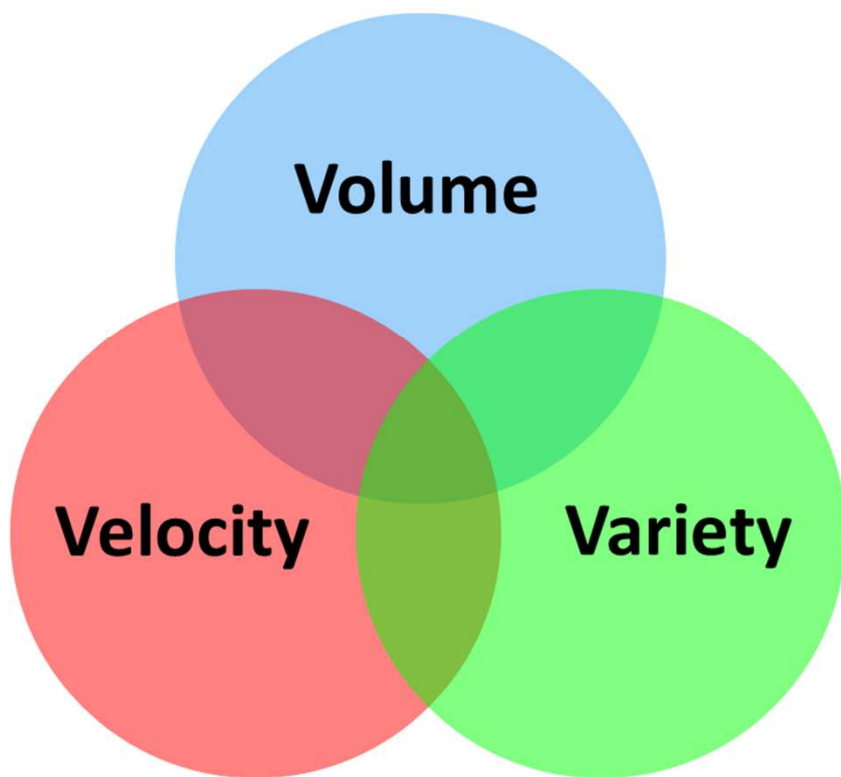


大数据系统与大规模数据分析

大数据运算系统 (2)



陈世敏

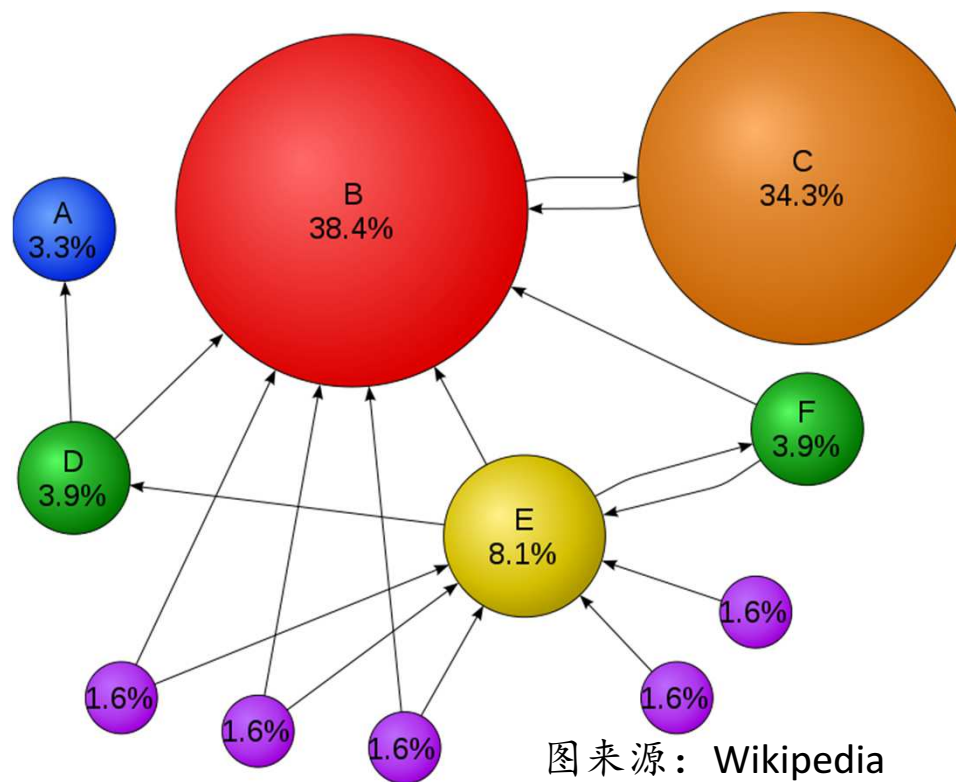
中科院计算所
计算机体系结构
国家重点实验室

©2015-2017 陈世敏

Outline

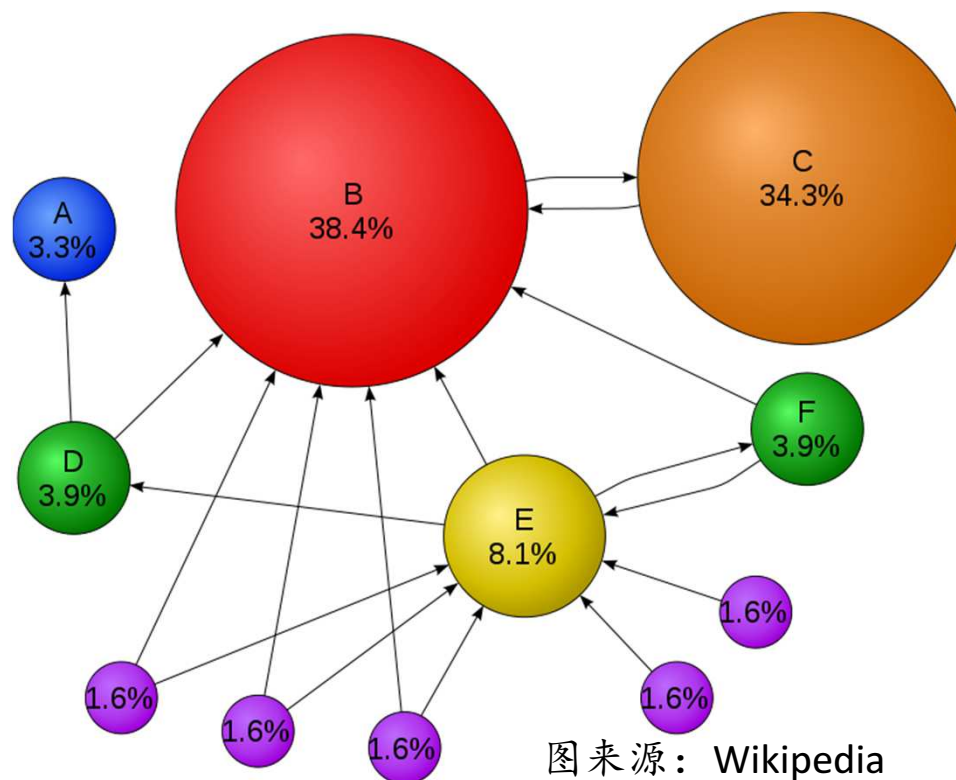
- 同步图计算系统
 - 图算法
 - 同步图计算
 - 图计算编程
 - 系统实现
- 异步图运算系统

图算法举例：PageRank



- Google用于对网页重要性打分的算法
- 上图简单示意了PageRank在一个图上的运行结果
 - 顶点：网页
 - 边：超链接

图算法举例：PageRank



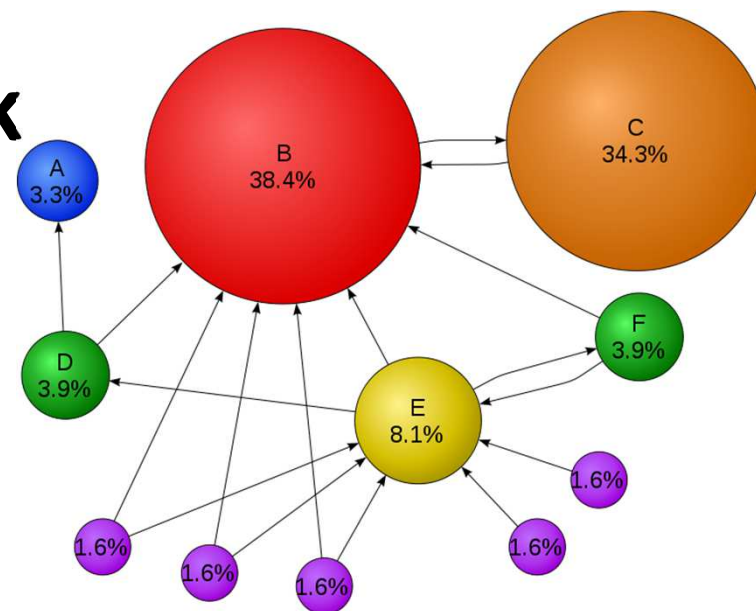
如果没有这种随机跳转，进入A,B,C后就出不来了

- 用户浏览一个网页时，有85%的可能性点击网页中的超链接，有15%的可能性转向任意的网页
 - PageRank算法就是模拟这种行为
 - $d=85\%$ (damping factor)

图算法举例：PageRank

- $$R_u = \frac{1-d}{N} + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

- R_v : 顶点v的PageRank
- L_v : 顶点v的出度（出边的条数）
- $B(u)$: 顶点u的入邻居集合
- d: damping factor
- N: 总顶点个数



图来源：Wikipedia

• 计算方法

- 初始化：所有的顶点的PageRank为 $\frac{1}{N}$
- 迭代：用上述公式迭代直至收敛

图算法举例：PageRank

- $R_u = \frac{1-d}{N} + d \sum_{v \in B(u)} \frac{R_v}{L_v}$

问题：N非常大时，数据精度可能不够？

- $NR_u = 1 - d + d \sum_{v \in B(u)} \frac{NR_v}{L_v}$

- 设 $R'_u = NR_u$

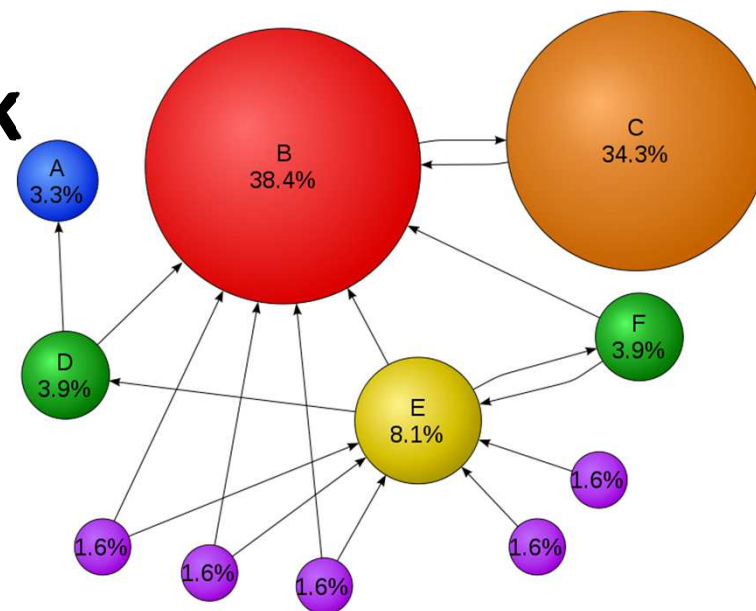
- R'_u 初始化为 1

- $R'_u = 1 - d + d \sum_{v \in B(u)} \frac{R'_v}{L_v}$

图算法举例：PageRank

- $R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$

- R_v : 顶点v的PageRank*N
- L_v : 顶点v的出度（出边的条数）
- $B(u)$: 顶点u的入邻居集合
- d: damping factor
- N: 总顶点个数



图来源：Wikipedia

• 计算方法

- 初始化：所有的顶点的PageRank为**1**
- 迭代：用上述公式迭代直至收敛

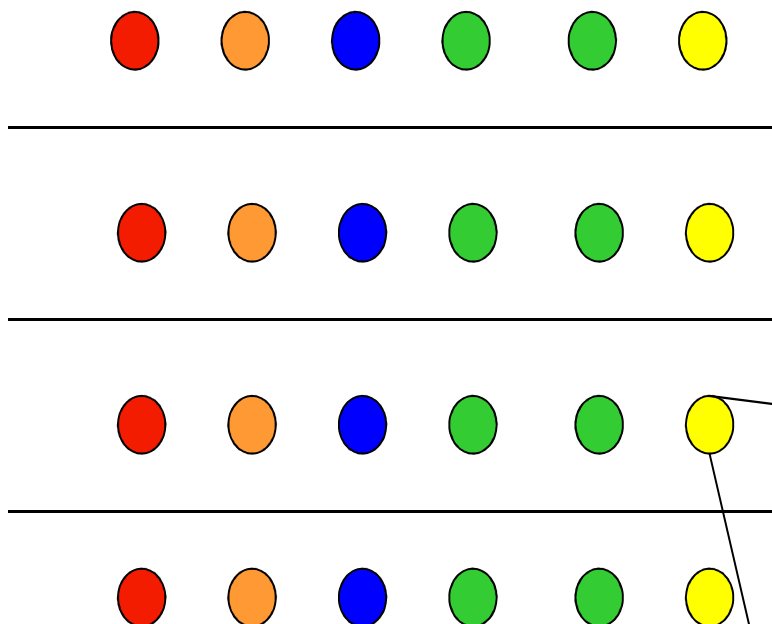
同步图运算系统

- “*Pregel: a system for large-scale graph processing.*”
Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, et al.
SIGMOD 2010.
- 开源实现: Apache Giraph, Apache Hama
- 我们的实现: GraphLite

图计算模型

运算
分成
多个
超步

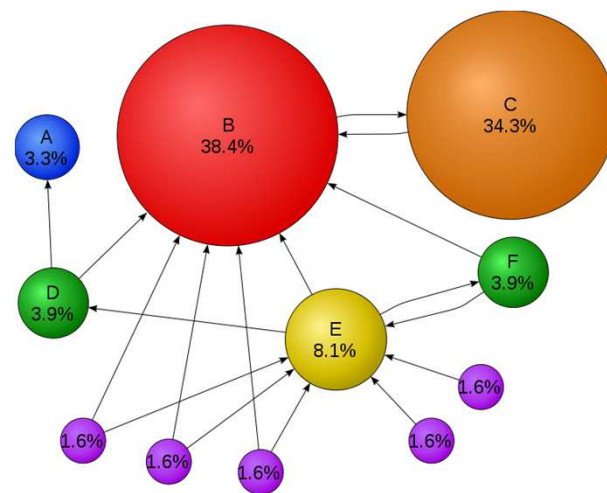
超步内，并行执行每个顶点



超步间全局同步

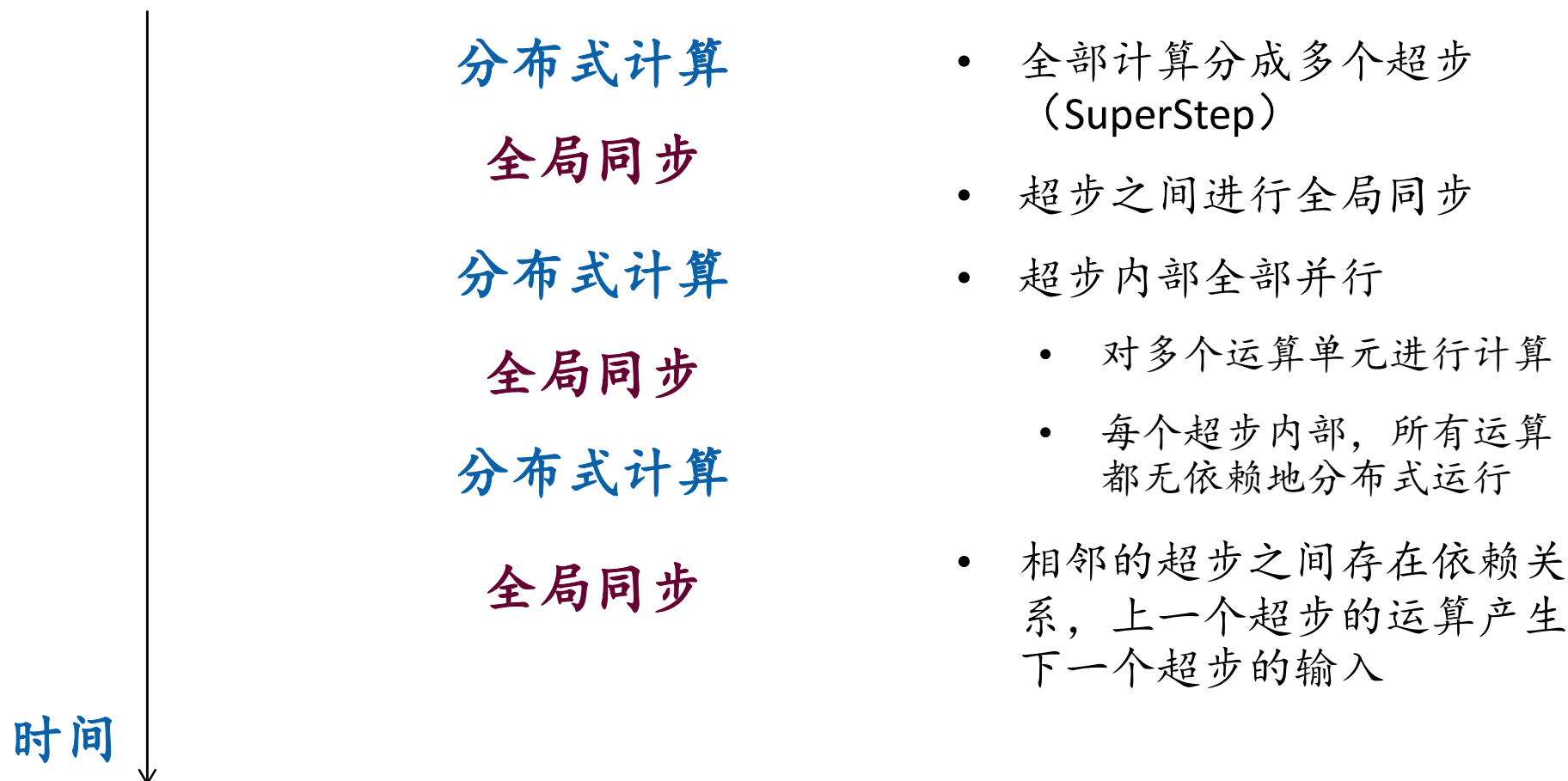
顶点算法通常步骤

- 1) 接收上个超步发出的 in-neighbor 的消息
- 2) 计算当前顶点的值
- 3) 向 out-neighbor 发消息

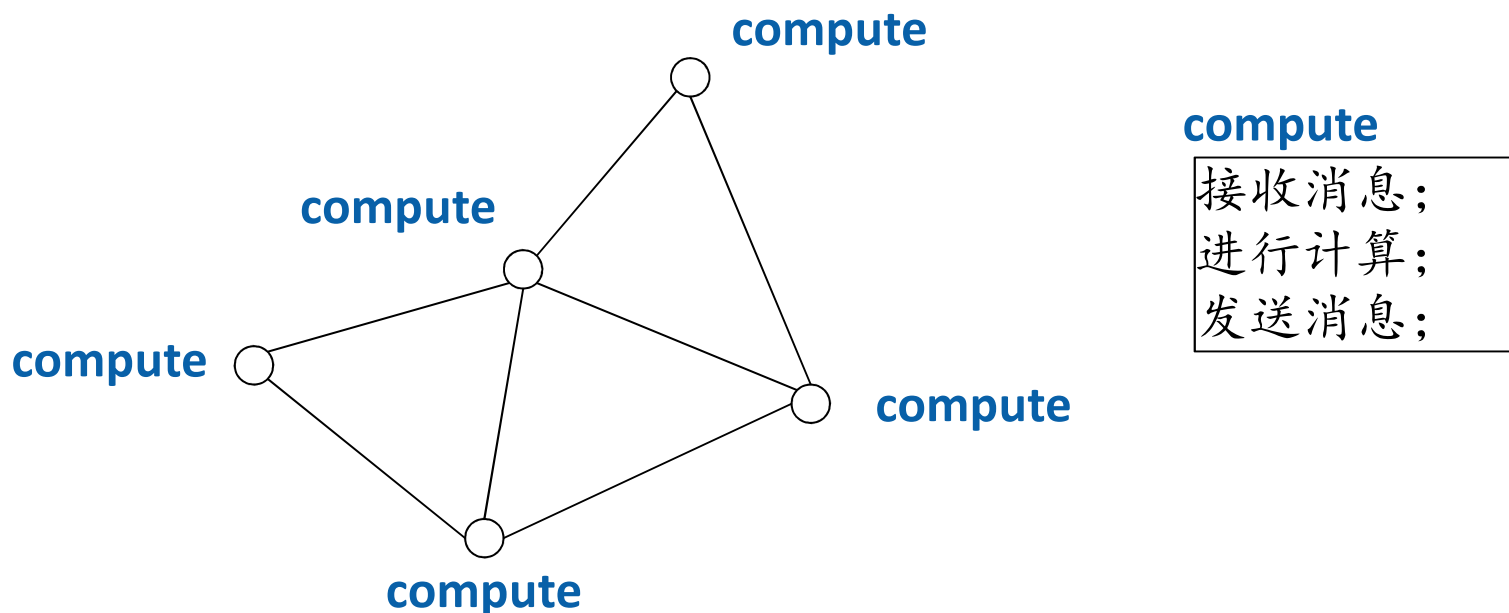


特点1: BSP模型

- BSP: Bulk Synchronous Processing

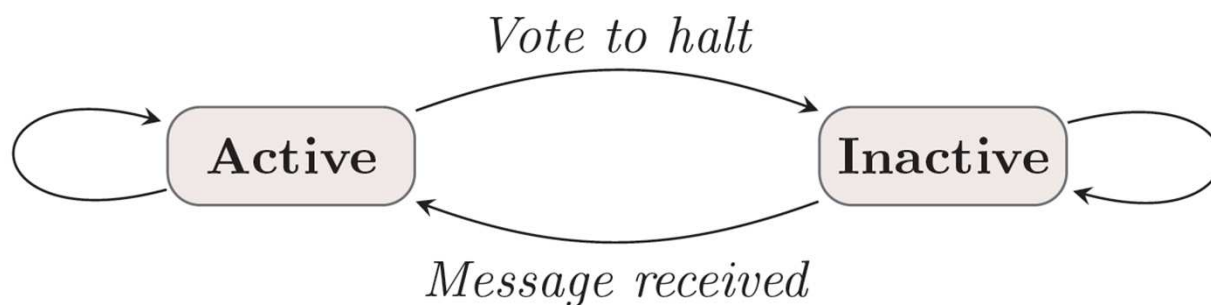


特点2：基于顶点的编程模型



- 每个顶点有一个value
- 顶点为中心的运算
 - 程序员可以实现一个Compute函数
 - 在每个超步中，同步图系统对每个顶点调用一次Compute
 - Compute通常接收消息，计算，然后发送消息

图运算如何结束？



- 顶点的两种状态

- 活跃态Active：图系统只对活跃顶点调用compute
 - 顶点初始状态都是活跃态
- 非活跃态Inactive：compute调用Vote to halt时，顶点变为非活跃态
 - 注意：非活跃的顶点也可以重新变得活跃

- 上图是顶点状态的转化图

- 当所有的顶点都处于非活跃状态时，图系统结束本次图运算

GraphLite

- 我们下面以GraphLite为例介绍同步图编程
- GraphLite实现了Pregel论文中定义的API
- GraphLite是C/C++实现的


<https://github.com/schencoding/GraphLite>

GraphLite编程过程

- 实现class Vertex的一个子类
- class Vertex中有两类函数
 - 1) 图计算程序员**需要实现的**：Compute()
 - 2) **系统提供的**，可以在Compute中调用的：
例如：getValue(), mutableValue(), getOutEdgeIterator(),
sendMessageTo(), sendMessageToAllNeighbors(),
voteToHalt()

Class Vertex

Vertex Value Type Edge Value Type Message Value Type



```
template<typename V, typename E, typename M>
class Vertex : public VertexBase {

    ...    ...

}
```

举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
class PageRankVertex: public Vertex<double, double, double>
{
    public:
        void compute(Messageliterator* msgs) { ... }
}
```

实现一个Vertex的子类

主要实现compute()函数

顶点、边和发送的消息的类型全为double

系统提供的函数 (1)

```
public: // methods provided by the system  
    const V & getValue();  
    V * mutableValue();
```

- 获得当前Vertex Value
 - getValue用于读
 - mutableValue用于修改

系统提供的函数 (2)

```
OutEdgeIterator getOutEdgeIterator();  
void sendMessageTo(const int64_t& dest_vertex ,  
                   const M & msg);  
void sendMessageToAllNeighbors(const M & msg);
```

- 发送消息给邻居顶点

- 每个顶点有唯一的ID: int64_t dest_vertex
- 如果发送给邻居的消息都相同, 那么可以用 sendMessageToAllNeighbors()
- 如果发给不同邻居的消息不同, 那么使用 getOutEdgeIterator() 得到 OutEdgeIterator, 然后可以依次访问邻边, 用 sendMessageTo() 发消息

系统提供的函数 (3)

```
void voteToHalt();
```

```
const int64_t & vertexID() const;
```

```
int superstep() const;
```

```
int getVSize() { return sizeof(V); }
```

```
int getESize() { return sizeof(E); }
```

- `voteToHalt()`
- `superstep()` 获取当前超步数：从0开始计数
- 设置Vertex Value和Edge Value类型的字节数

系统提供的函数（4）

```
void accumulate(const void * p, int agg_id);  
const void* getAggregate(int agg_id);
```

- 全局的统计量

需要实现的函数

```
virtual void compute( MsgIterator * msgs )=0;
```

举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
class PageRankVertex: public Vertex<double, double, double>
{
    public:
        void compute(MessageIterator* msgs) { ... }
}
```

顶点、边和发送的消息的类型全为double

举例：PageRank实现

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
void compute( MsgIterator * msgs )
{
    double val;
    if ( superstep() == 0 ) {
        val = 1.0; // initial value
    }
    else {
        // 正常执行PageRank迭代，计算val

    }
    // set new pagerank value and propagate
    *mutableValue() = val;
    int64_t n = getOutEdgeIterator().size();
    sendMessageToAllNeighbors( val / n );
}
```

举例：PageRank实现

else {

$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
// compute pagerank
```

```
double sum= 0.0;
```

```
for (; !msgs->done(); msgs->next()) {
```

```
    sum += msgs->getValue();
```

```
}
```

```
val = 0.15 + 0.85 * sum;
```

```
}
```


举例：PageRank实现

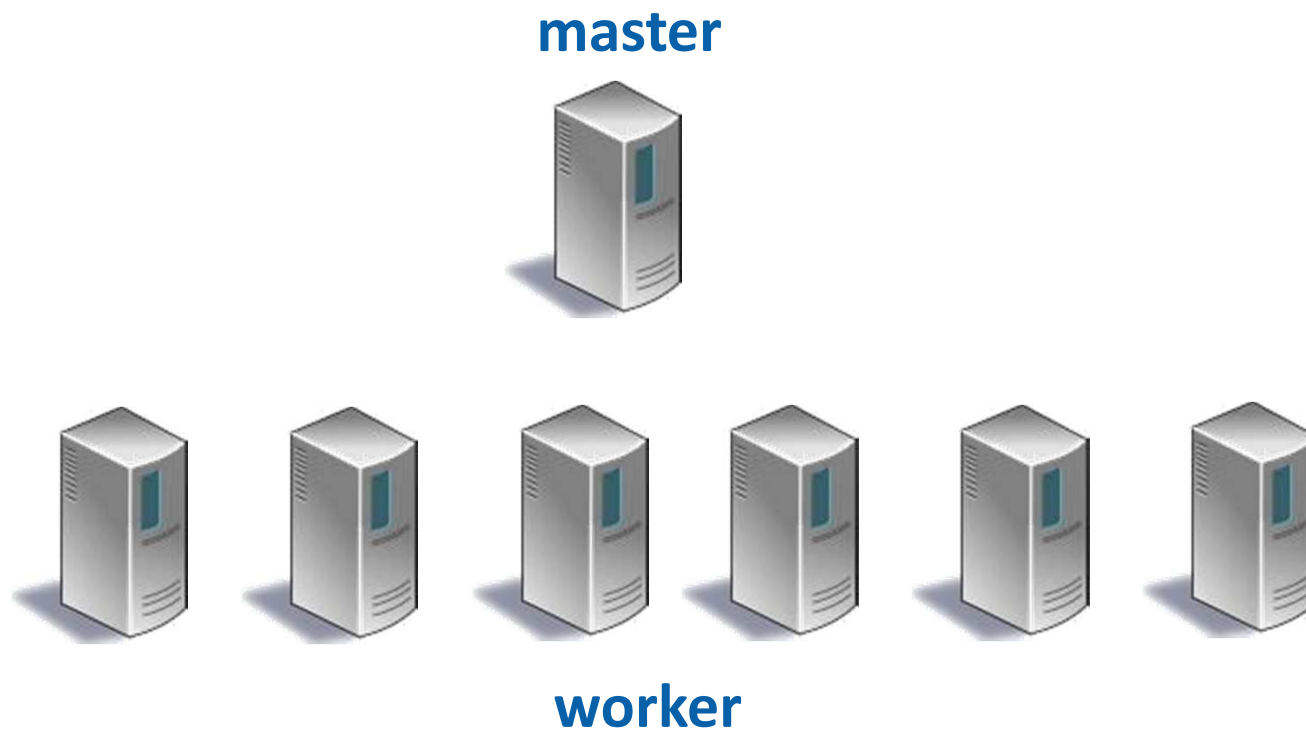
$$R_u = 1 - d + d \sum_{v \in B(u)} \frac{R_v}{L_v}$$

```
else {  
    // check if converged  
    if (superstep() >= 2 &&  
        *(double *)getAggregate(AGGERR) < TH) {  
        voteToHalt(); return;  
    }  
    // compute pagerank  
    double sum= 0.0;  
    for (; !msgs->done(); msgs->next()) {  
        sum += msgs->getValue();  
    }  
    val = 0.15 + 0.85 * sum;  
    // accumulate delta pageranks  
    double acc = fabs(getValue() - val);  
    accumulate(&acc, AGGERR);  
}
```

其它部分

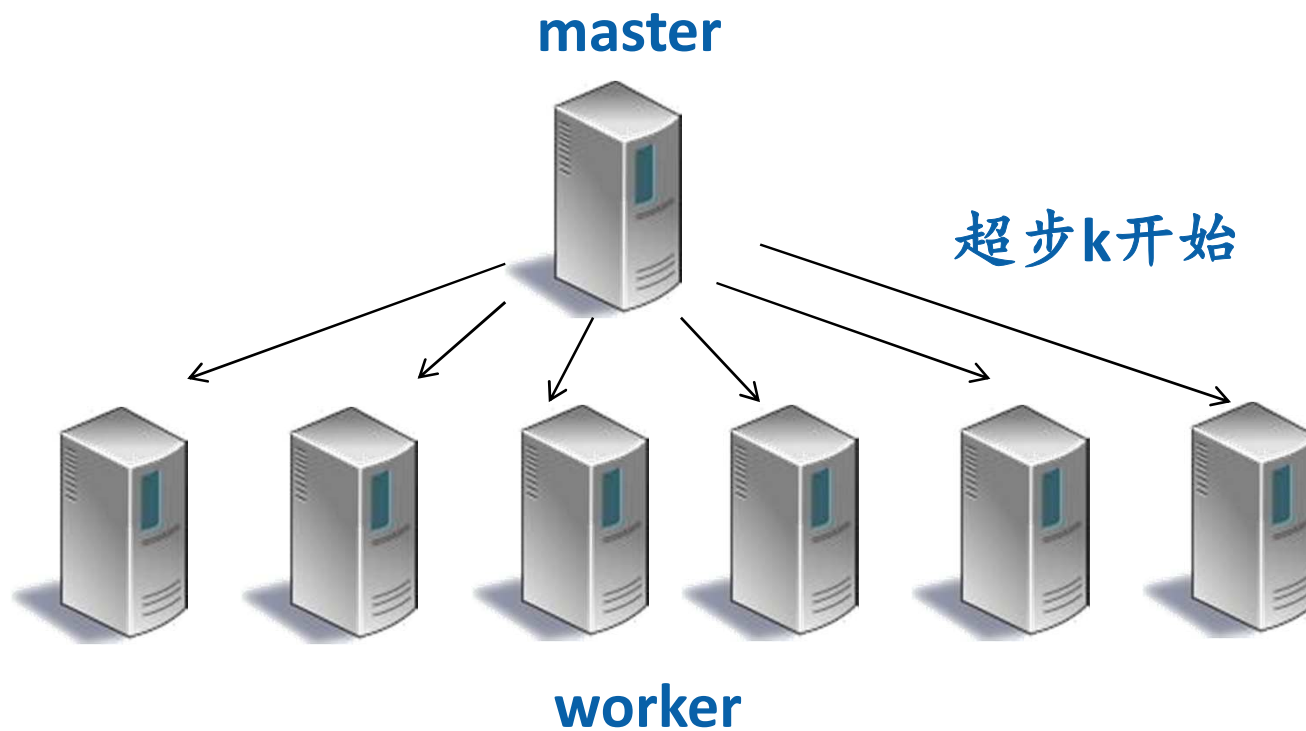
- 初始化
- InputFormatter
- OutputFormatter
- Aggregator

同步图运算系统的系统架构

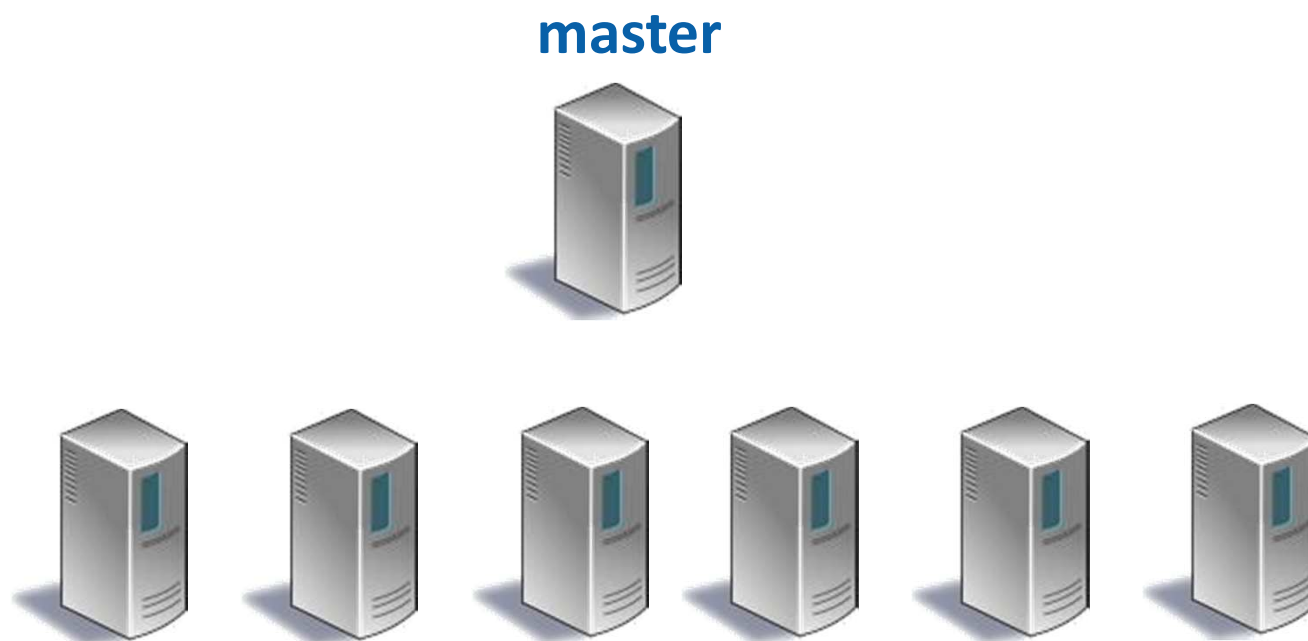


- 每个worker对应一个graph partition
- 例如: hash partition
 - $\text{Partition id} = \text{hash}(\text{vertex_id}) \% \text{WorkerNumber}$

超步开始

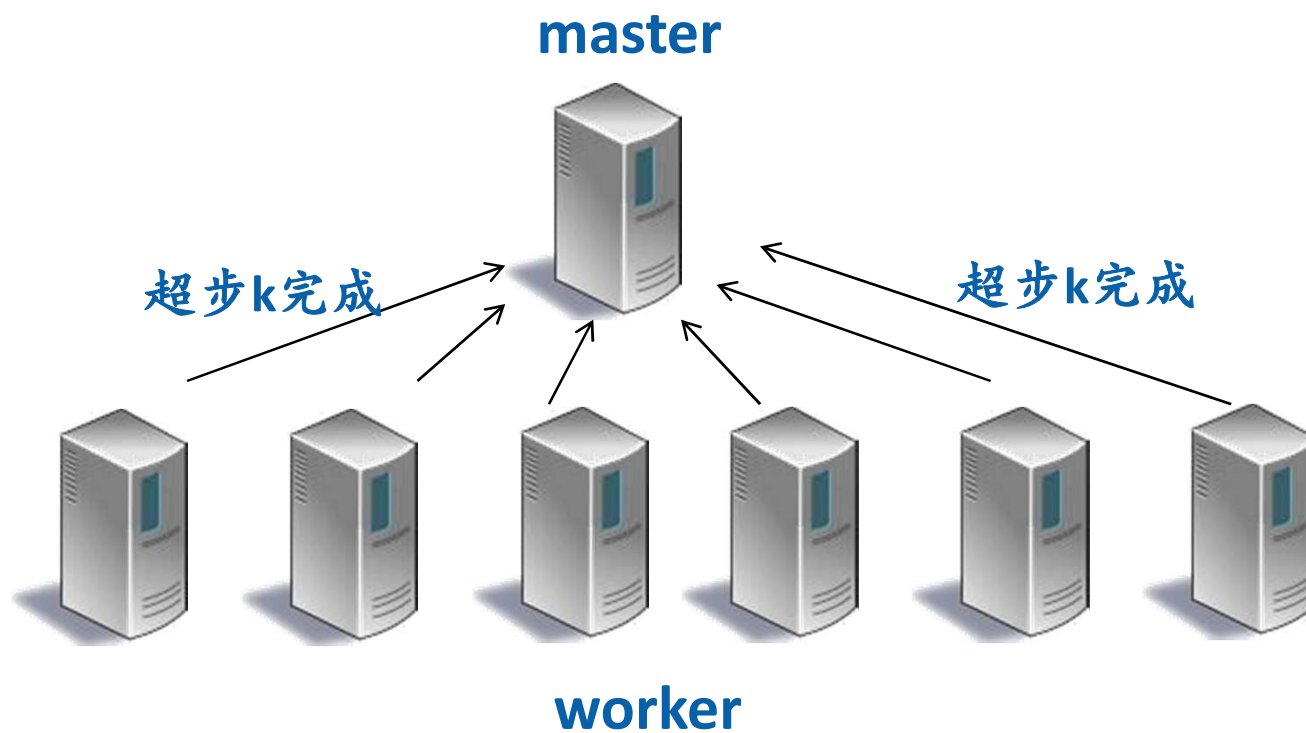


超步计算进行中

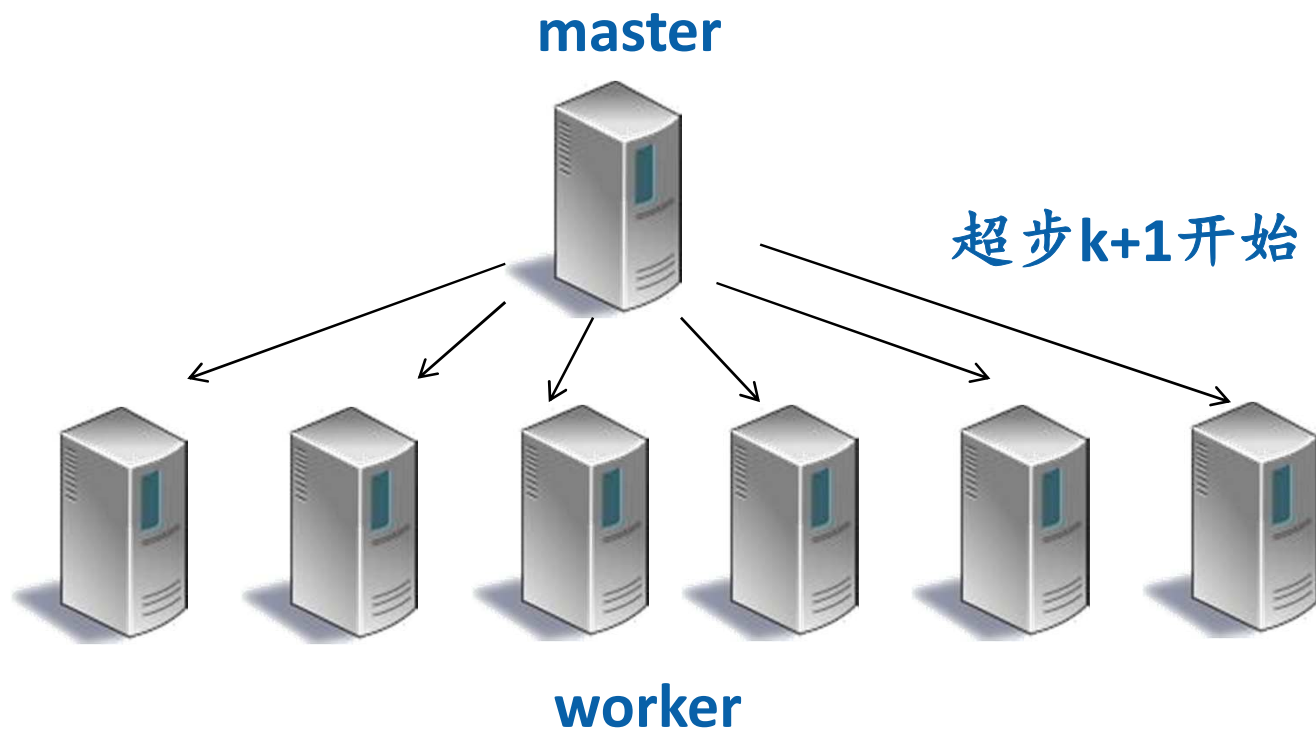


每个worker进行本地的计算，为本partition的每个顶点调用compute，收集顶点发送的信息，并发向对应的worker

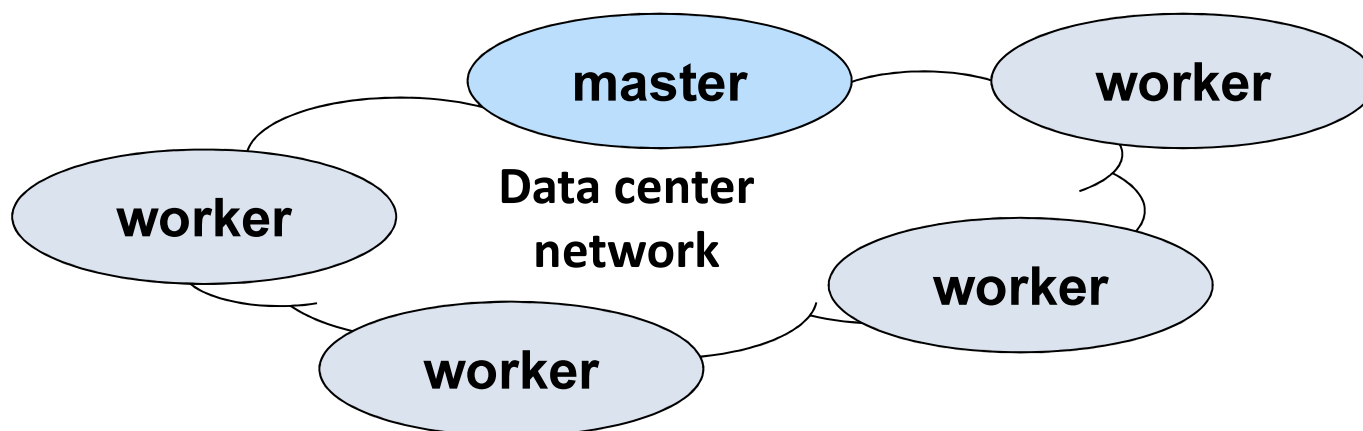
超步结束



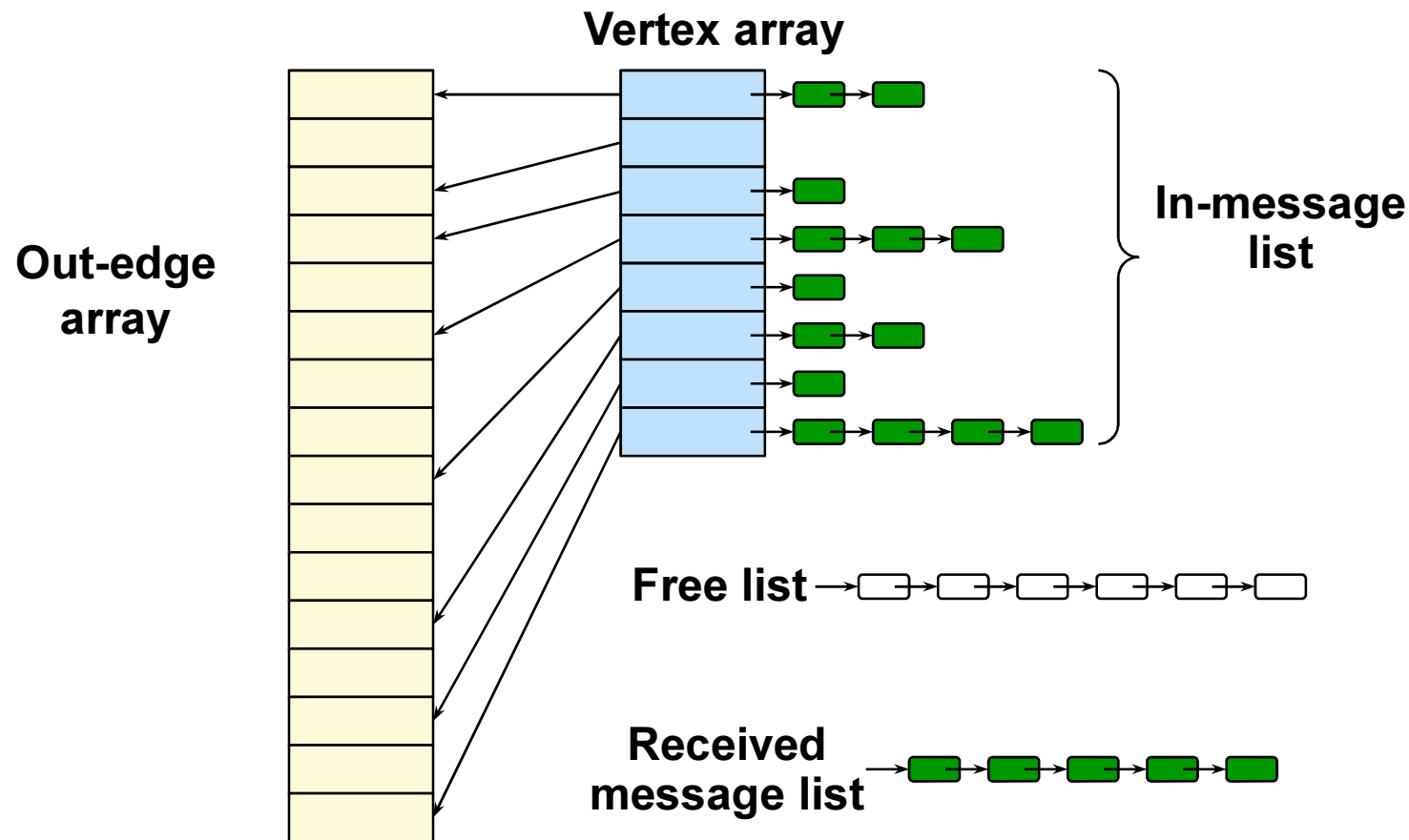
超步开始



GraphLite



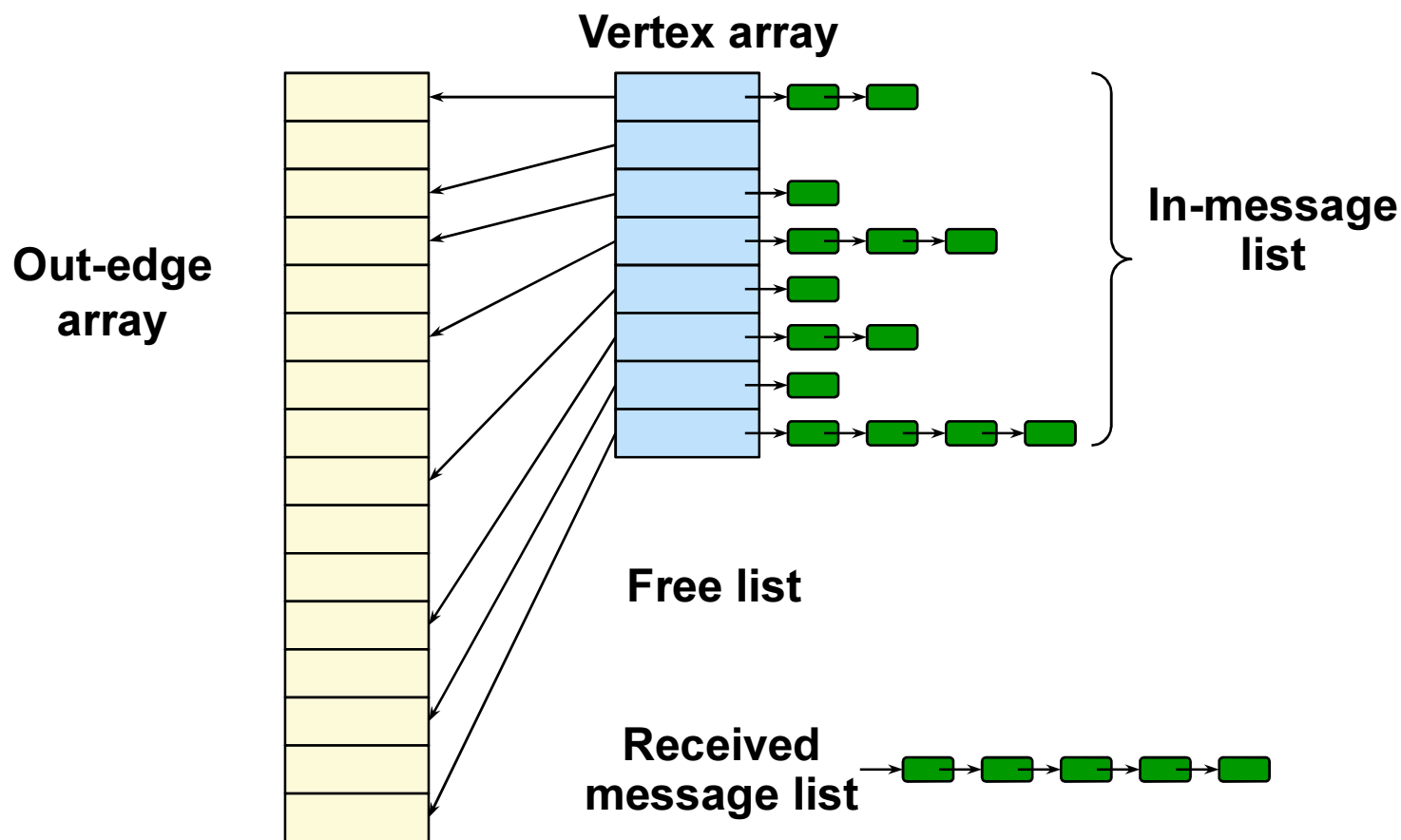
GraphLite Worker



Message: (source ID, target ID, message value, ptr)

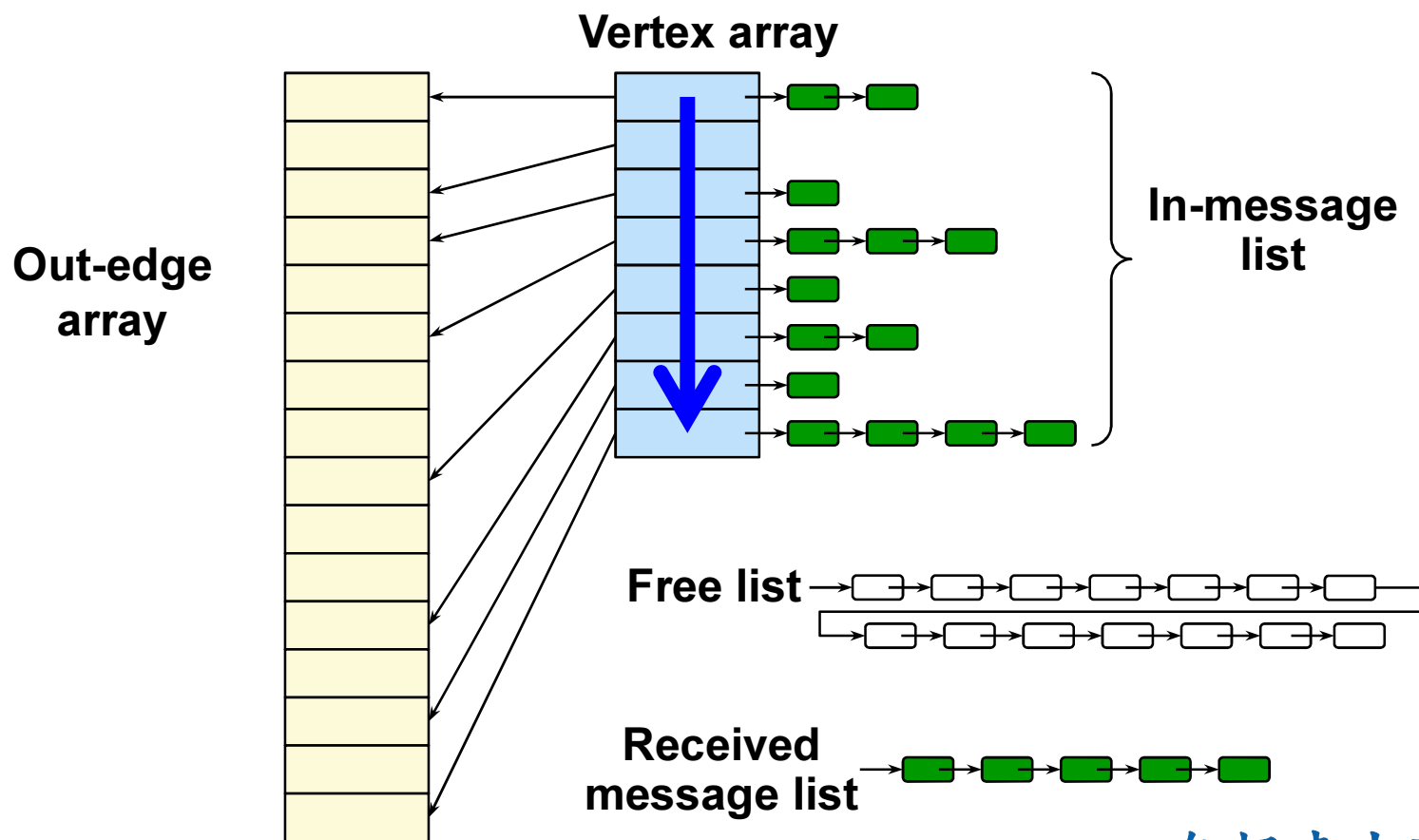
超步开始：分发message

把Received message list
中的消息放入接收顶点的
in-message list



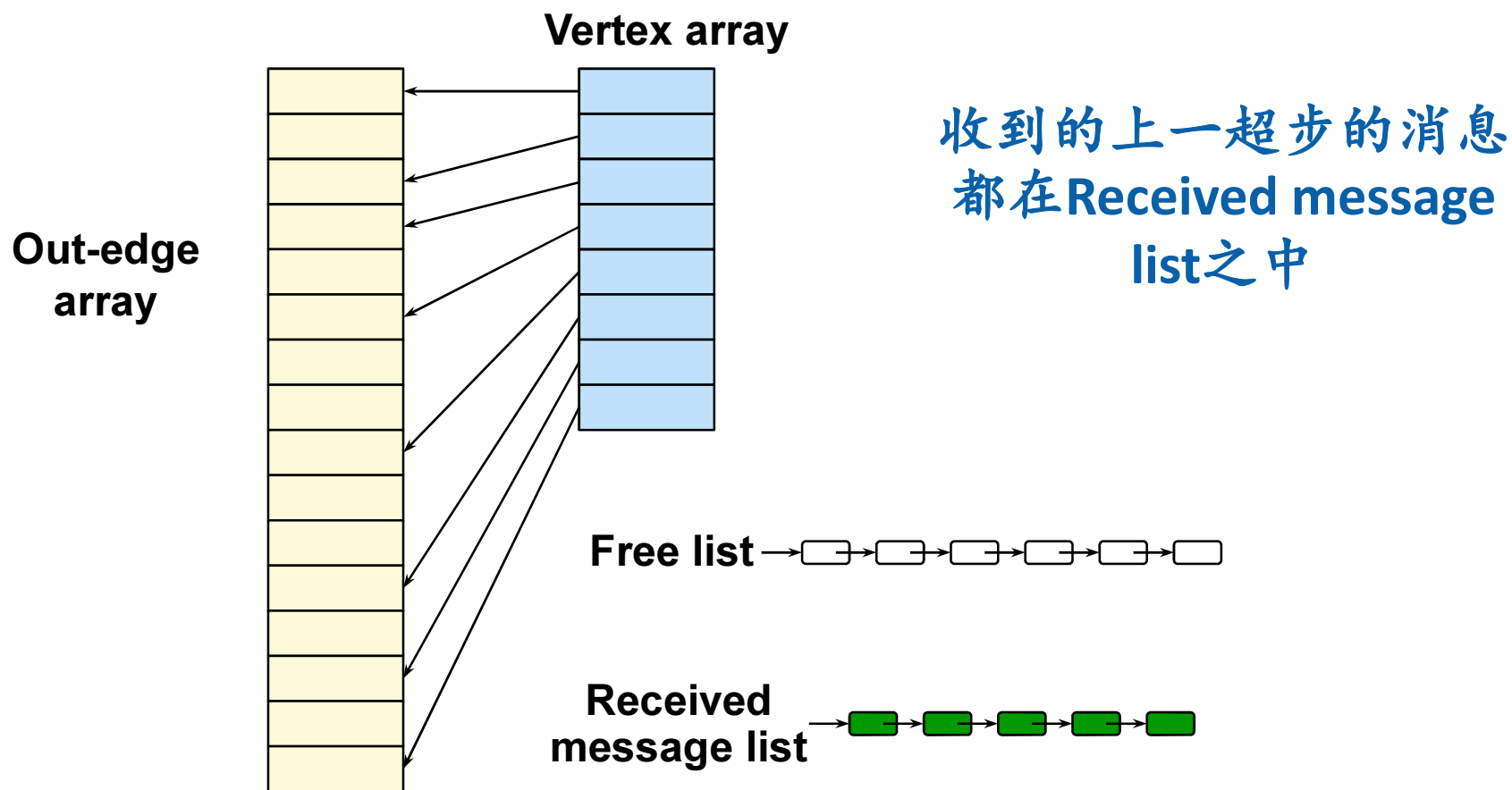
Message: (source ID, target ID, message value, ptr)

超步计算中：依次访问Vertex, 调用Compute



在超步中可能收到消息

超步结束时



Message: (source ID, target ID, message value, ptr)

Aggregator全局统计量

- 第0个超步内
 - 每个Worker分别进行本地的统计
- 超步间，全局同步时
 - Worker把本地的统计发给master
 - Master进行汇总，计算全局的统计结果
 - Master把全局的统计结果发给每个Worker
- 下一个超步内
 - Worker从Master处得到了上个超步的全局统计结果
 - Compute就可以访问上一超步的全局统计信息了
 - 继续计算本超步的本地统计量

同步图运算系统小结

- 基于BSP模型实现同步图运算
- 运算在内存中完成
- 容错依靠定期地把图状态写入硬盘生成检查点
 - 在一个超步开始时，master可以要求所有的worker都进行检查点操作
- 可以比较容易地表达一些图操作

Outline

- 同步图运算系统
- 异步图运算系统
 - 数据模型
 - 计算过程

异步图运算系统

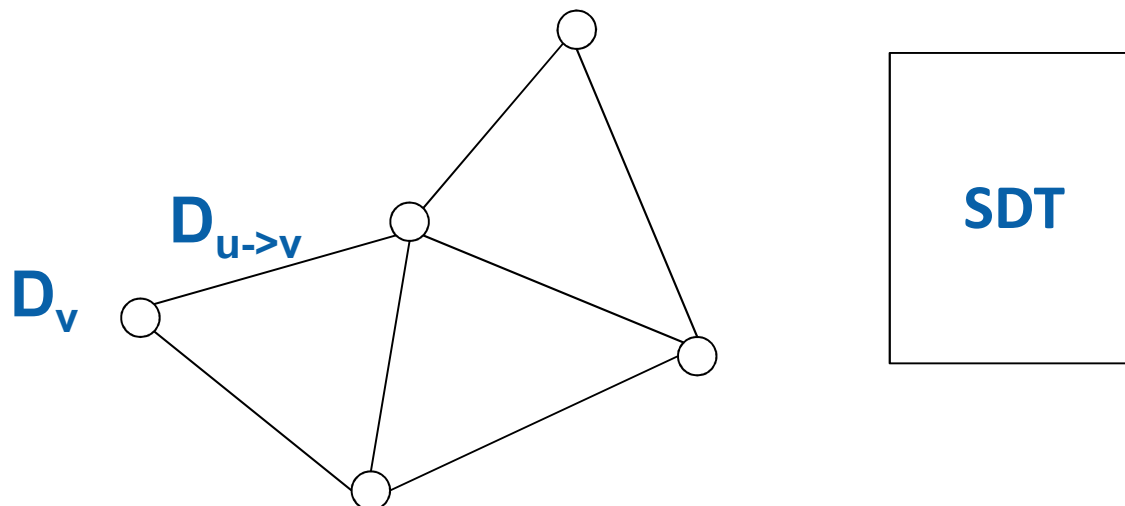
- 许多机器学习算法可以用图运算来实现
- 但是同步图运算需要进行多次的迭代
- 思路：异步图运算
 - 允许不同顶点有不同的更新速度
 - 一个顶点的更新，它的邻居顶点立即可见，而不是等到下一个超步开始
 - 从而可以更快速地收敛
 - 注意：许多机器学习算法没有“精确”的结果
 - 异步计算收敛的结果和同步计算收敛的结果可能是不一样的

异步图运算系统 GraphLab

- “*GraphLab: A New Framework For Parallel Machine Learning*”. Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein. **UAI 2010**.
- “*Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud*.” Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin and Joseph M. Hellerstein. **PVLDB 2012**.
- “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.” Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. **OSDI 2012**.

数据模型

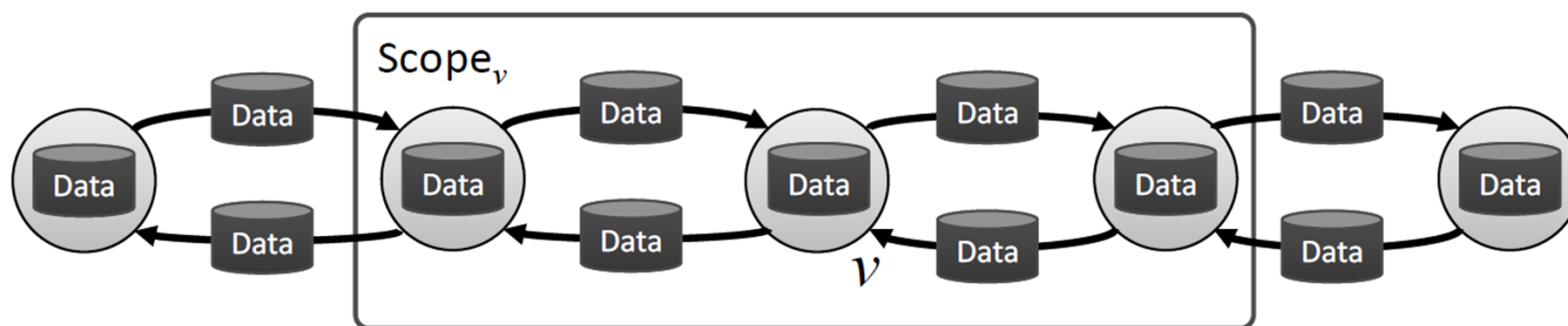
- Data graph $G=(V, E)$
 - 每个顶点可以有数据 D_v
 - 每条边可以有数据 $D_{u \rightarrow v}$
- 全局数据表(SDT, shared data table)
 - $SDT[key] \rightarrow value$
 - 可以定义全局可见的数据



顶点计算

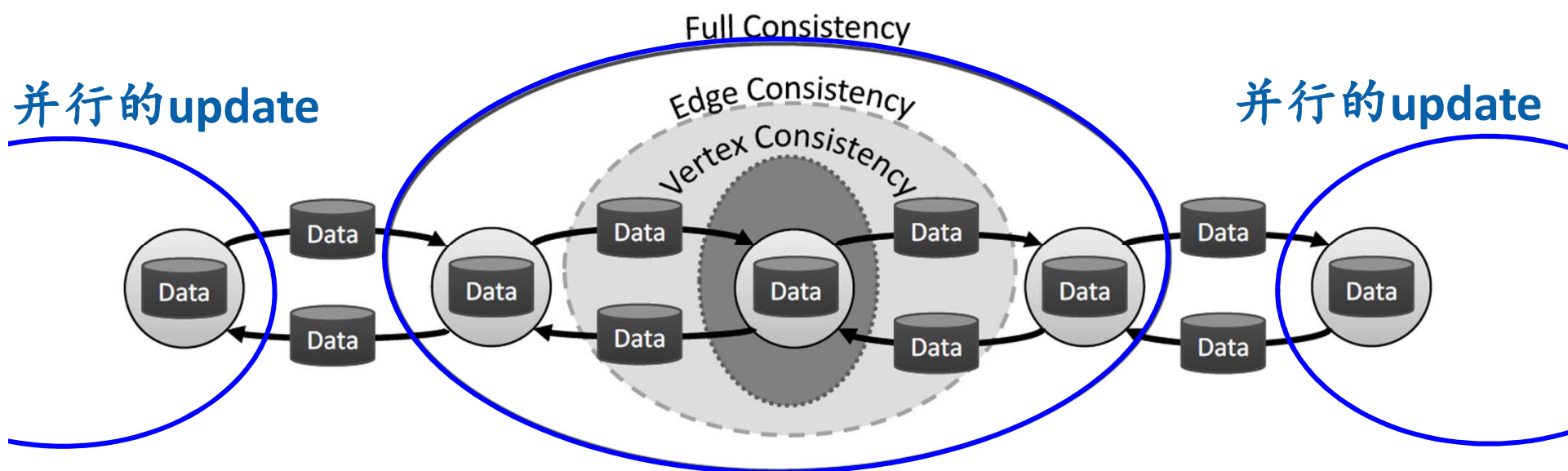
- $D_{Scope_v} = \text{update}(D_{Scope_v}; \text{SDT})$

- update 类似 Pregel compute, 是程序员定义的顶点运算
- $Scope_v$ 是顶点运算涉及的范围
 - 包括顶点 v , v 的相邻边, 和 v 的相邻顶点
- 运算是 **直接访问内存** 进行的
- update 直接修改顶点和边上的数据, **修改立即可见**, 而不是下个超步才可见



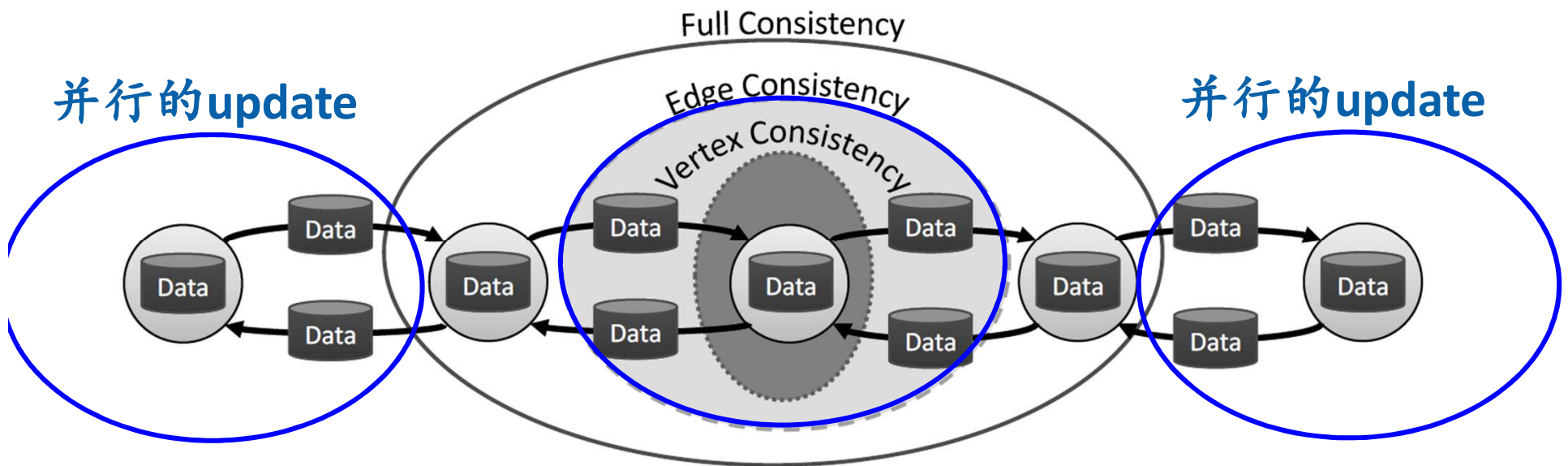
三种一致性模型

- Full consistency: 系统保证在update执行过程中，没有任何其它函数会访问Scope_v
 - 这种模型对并行执行的限制最大
 - 但是任何update都能够正确执行



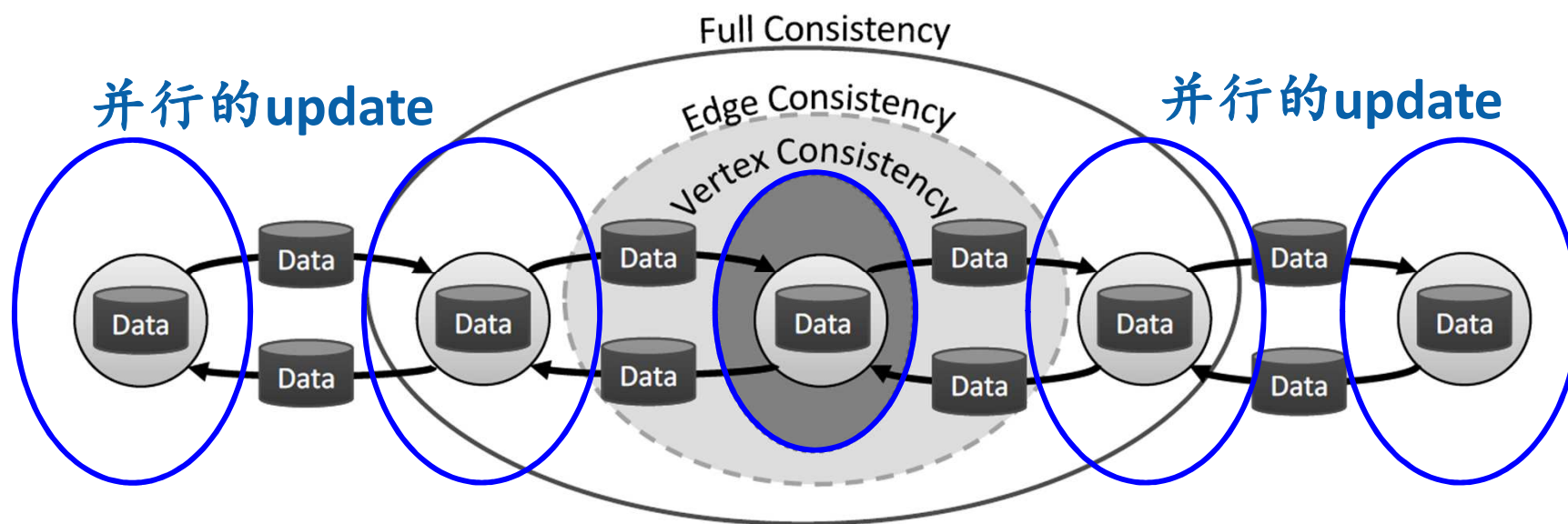
三种一致性模型

- Edge consistency: 系统保证在update执行过程中，没有任何其它函数会访问v的Edge和v本身
 - update不能写邻居顶点的数据，但是可以读邻居顶点的数据



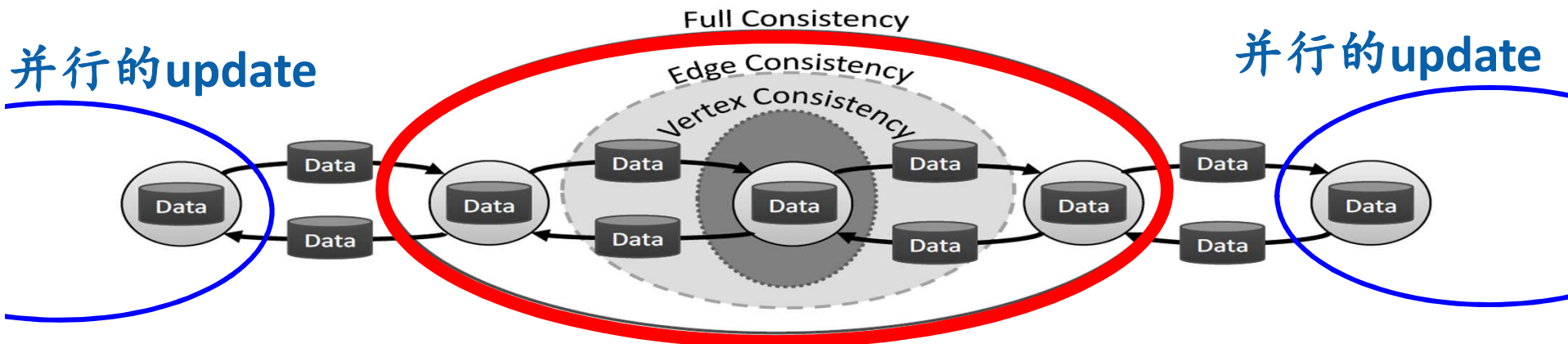
三种一致性模型

- Vertex consistency: 系统保证在update执行过程中，没有任何其它函数会访问v本身
 - Update只能够读写本顶点的数据，和读v邻边的数据



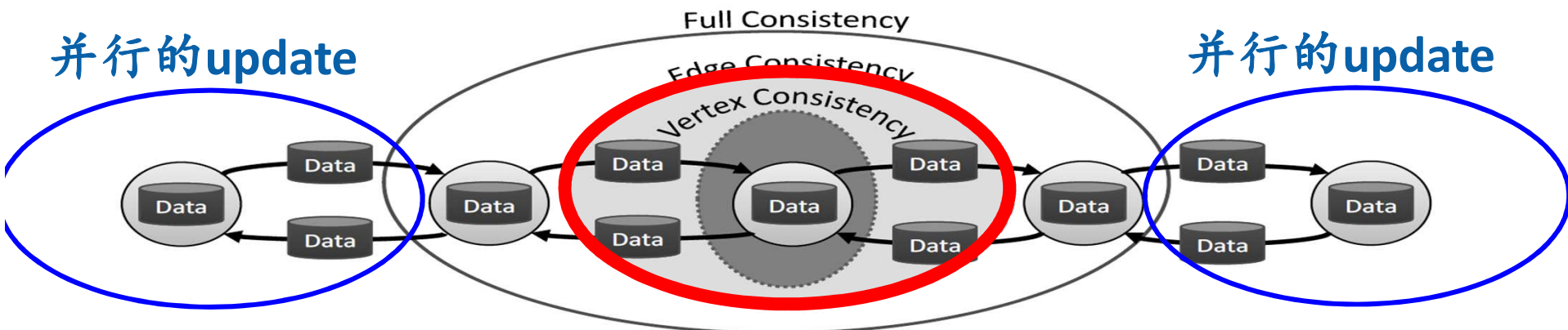
并行的update

并行的update



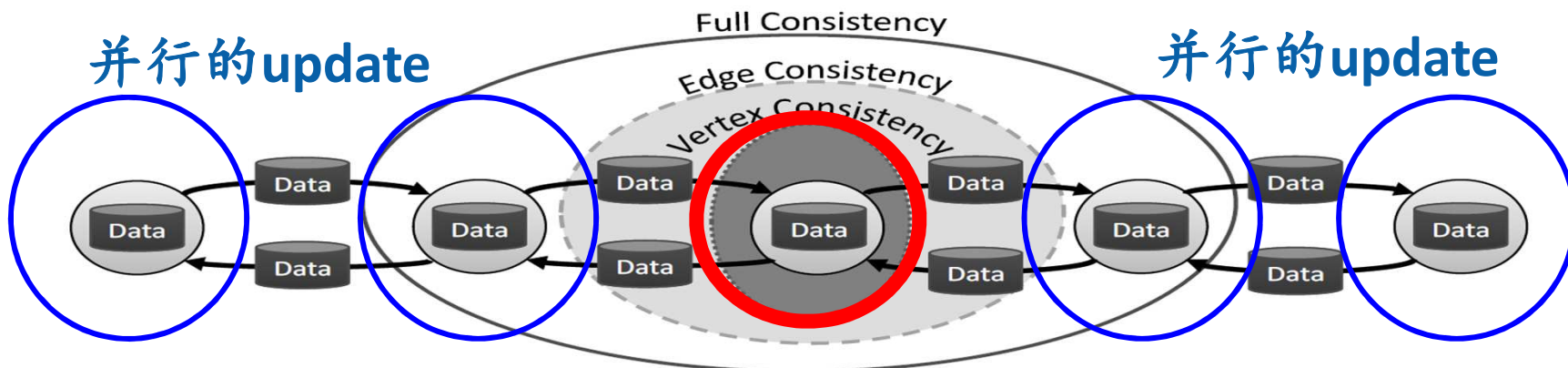
并行的update

并行的update



并行的update

并行的update



Scheduling

- 按照什么顺序访问顶点调用update?
- 例如:
 - Synchronous scheduler: 类似同步图运算
 - Round-robin scheduler: 顺序计算每个顶点, 下一个顶点可以看到前面的计算结果
 - FIFO scheduler: update中调用AddTask创建新的Task, Task对应顶点的update, 创建的Task按照先进先出顺序执行
 - Prioritized scheduler: 创建Task并指定优先顺序

Sync计算

- 除了update计算，GraphLab还定义了一种全局的计算sync
- Sync类似在所有的顶点上计算一个aggregate
 - 程序员提供fold和apply函数，给定一个初始值initial_value和一个key
 - 系统执行下面的操作：

```
t = initial_value;
foreach v ∈ V {
    t = fold(v, t);
}
SDT[key] = apply(t);
```
- 例如：可以计算update运算是否已经收敛

GraphLab

- 以顶点为中心的计算
- 异步计算
 - 可以定义不同的scheduling策略
- 可以立即看到完成的计算结果
 - 共享内存方式编程，而不是消息传递方式
 - 需要一致性模型
 - Full, edge, vertex consistency模型
- 采用一种全局的aggregate机制帮助判断是否收敛

Outline

- 同步图运算系统
- 异步图运算系统
 - 数据模型
 - 计算过程

小结

- 同步图运算系统: Pregel, GraphLite
- 异步图运算系统: GrapLab