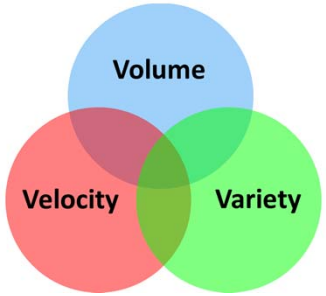
大数据系统与大规模数据分析

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

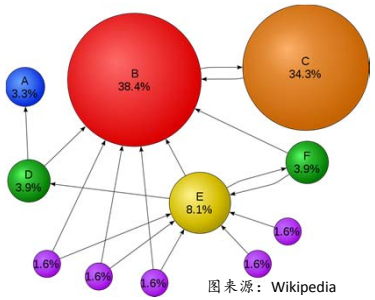
Outline

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

图(Graph)的概念

- G=(V, E)
  - V：顶点(Vertex)的集合
  - E：边(Edge)的集合
    - 边 e=(u,v), u∈V, v∈V
- 有向图（directed graph）
  - 边有方向
- 无向图
  - 边没有方向
  - 可以用有向图表达无向图：每条无向边➔2条有向边

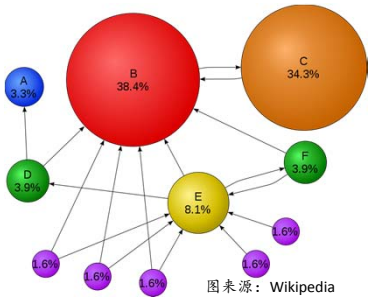
图算法举例：PageRank



图来源：Wikipedia

- 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



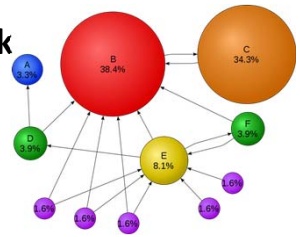
图来源：Wikipedia

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

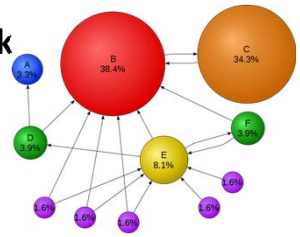
- $R_u = \frac{1-d}{N} + d\sum_{v \in B(u)} \frac{R_v}{L_v}$

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



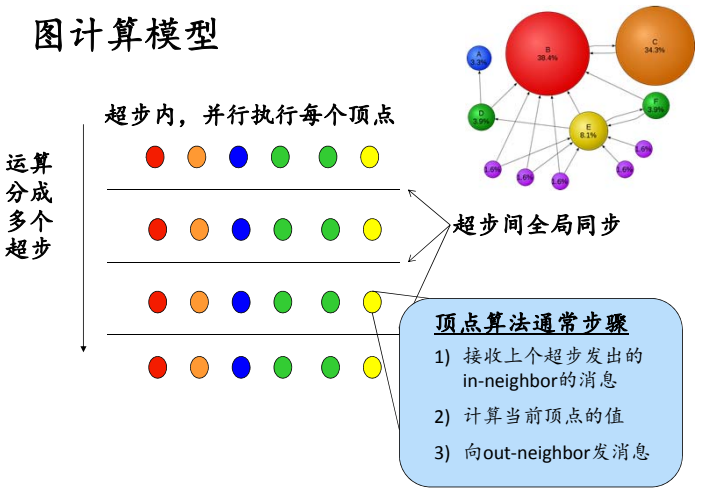
图来源：Wikipedia

- 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

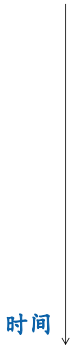
- "*Pregel: a system for large-scale graph processing.*" Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, et al. SIGMOD 2010.
- 开源实现: Apache Giraph, Apache Hama
- 我们的实现：GraphLite

图计算模型



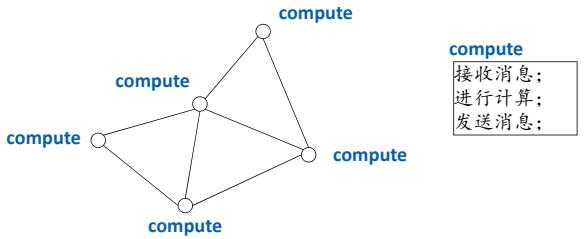
特点1：BSP模型

- BSP: Bulk Synchronous Processing



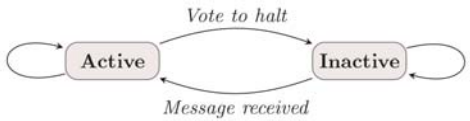
- 全部计算分成多个超步（SuperStep）
- 超步之间进行全局同步
- 超步内部全部并行
  - 对多个运算单元进行计算
  - 每个超步内部，所有运算都无依赖地分布式运行
- 相邻的超步之间存在依赖关系，上一个超步的运算产生下一个超步的输入

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Volt 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

图计算编程

- 数据
  - 顶点？
  - 边？
  - 消息？
- 运算
  - Compute?

GraphLite编程

- 继承class Vertex，实现一个子类
- 可以定义
  - 顶点数据、边数据、消息数据的类型
  - 实现Compute函数

Class Vertex

Vertex Value Type　Edge Value Type　Message Value Type

```
template<typename V, typename E, typename M>
class Vertex : public VertexBase {
  public:
    void compute(MessageIterator* msgs) { … }
}
```


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) { … }
}
```

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

GraphLite系统提供的函数

系统提供的，可以在Compute中调用的

- getValue(), mutableValue(),
- getOutEdgeIterator(), sendMessageTo(), sendMessageToAllNeighbors(),
- voteToHalt()，superstep()等
- accumulate()等

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

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

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

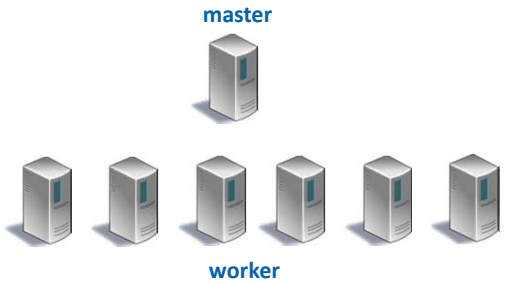
```
else {
   // compute pagerank
   double sum= 0.0;
   for (; !msgs->done(); msgs->next()) {
      sum += msgs->getValue();
   }
   val = 0.15 + 0.85 * sum;
}
```


举例：PageRank实现

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

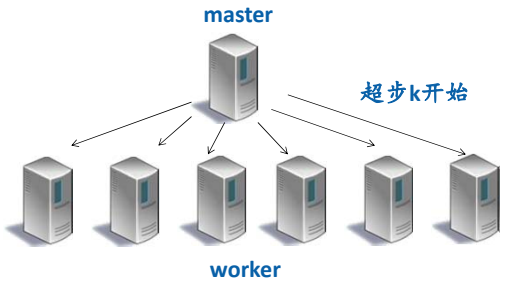
```
else {
  // check if converged
  if (superstep() >= 2 &&
      *(double *)getAggregate(AGGERR) < TH) {
    voteToHalt(); return;
  }
  // compute pagerank
  double sum= 0.0;
  for (; !msgs->done(); msgs->next()) {
     sum += msgs->getValue();
  }
  val = 0.15 + 0.85 * sum;
  // accumulate delta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


同步图运算系统的系统架构

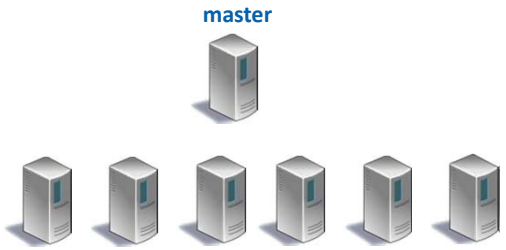


- 每个worker对应一个graph partition
- 例如：hash partition
  - Partition id = hash (vertex_id) % WorkerNumber

超步开始

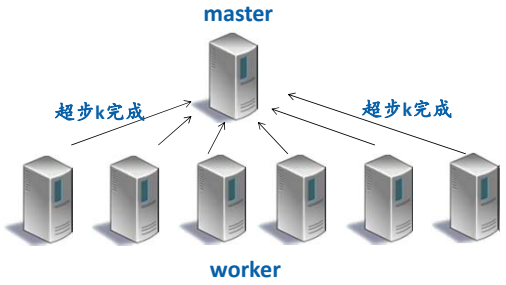


超步计算进行中

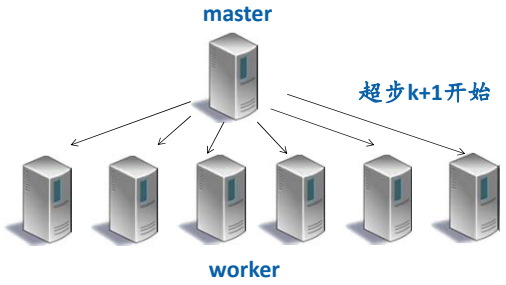


每个worker进行本地的计算，为本partition的每个顶点调用compute，收集顶点发送的信息，并发向对应的worker

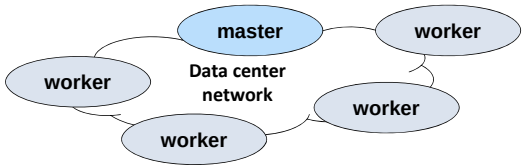
超步结束



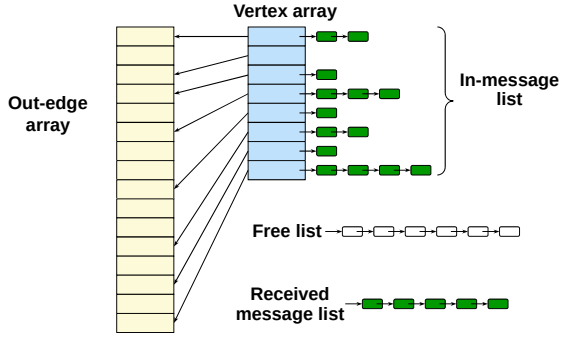
超步开始



GraphLite



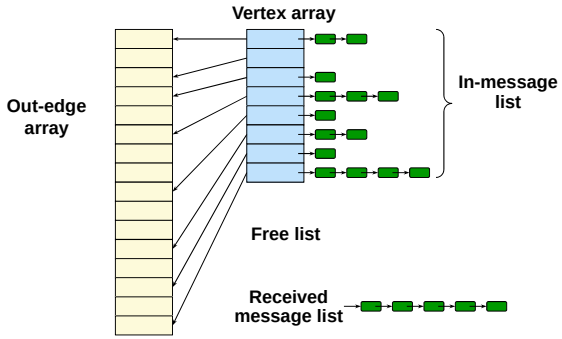
GraphLite Worker



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

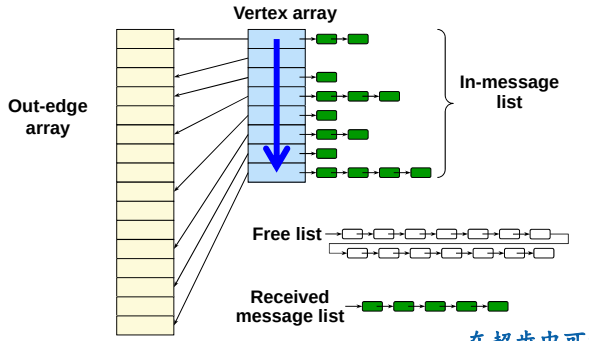
超步开始：分发message

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



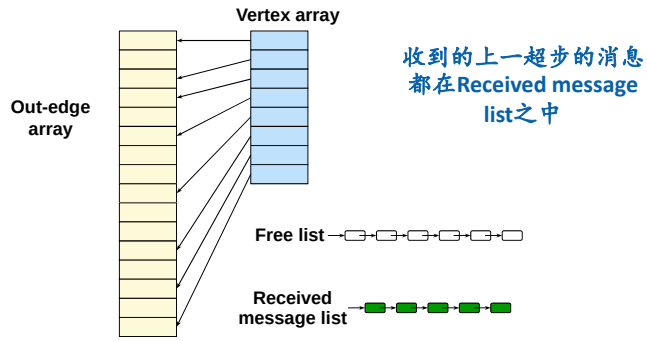
Message: (source ID, target ID, message value, ptr)

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



收到的上一超步的消息都在Received message list之中

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