基于状态更新传播的流式图计算系统设计与实现

段世凯
许利杰、王伟

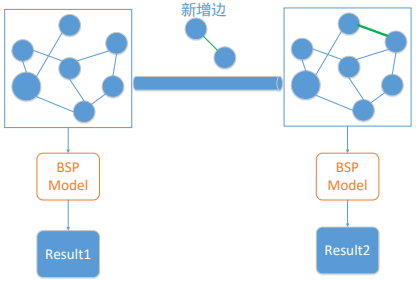
中国科学院软件研究所
软件工程技术研究开发中心

纲要

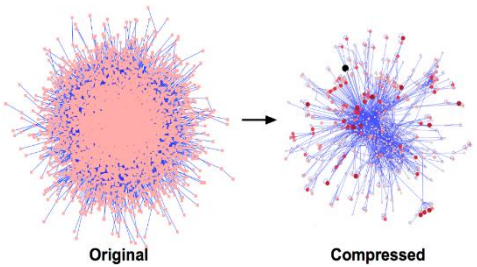
- 流式图计算研究现状
- 流式图算法特征分析
- 解决方案
- 实验结果
- 总结和下一步规划

流式图计算研究现状

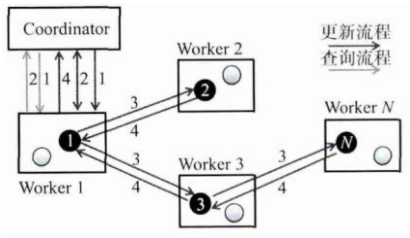
BSP模型



采样 & 概要



KineoGraph/IncGraph增量图计算系统



SpecGraph
并发更新
图计算系统

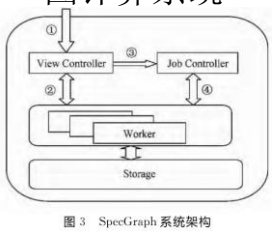


图 3 SpecGraph 系统架构

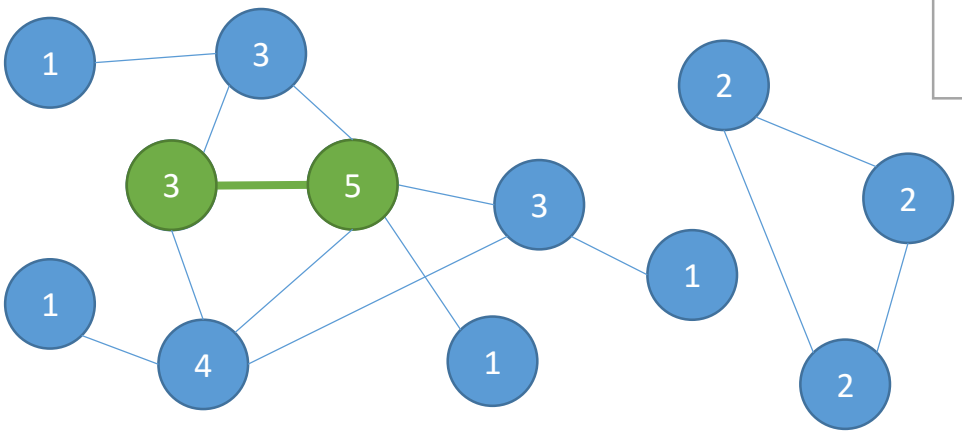
等待周期长 **准确率低** **串行更新** **表达能力差**

构建面向**流式图数据**的**精准计算**的**实时图计算**系统

- ✓ 满足实时计算要求 (<20ms)
- ✓ 有较高的准确率 (>99%)
- ✓ 采用并行更新方式
- ✓ 表达能力强(≈VC模型表达能力)

流式图算法特征分析

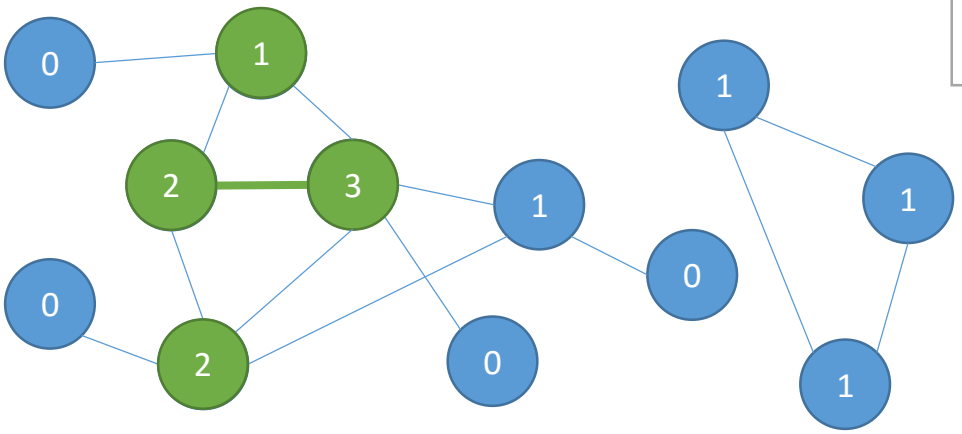
举例：统计各个节点的度



- 只影响增加边所在的两个节点
- 这两个节点只需计算一次
- 计算结果与各个节点的计算顺序无关

流式图算法特征分析

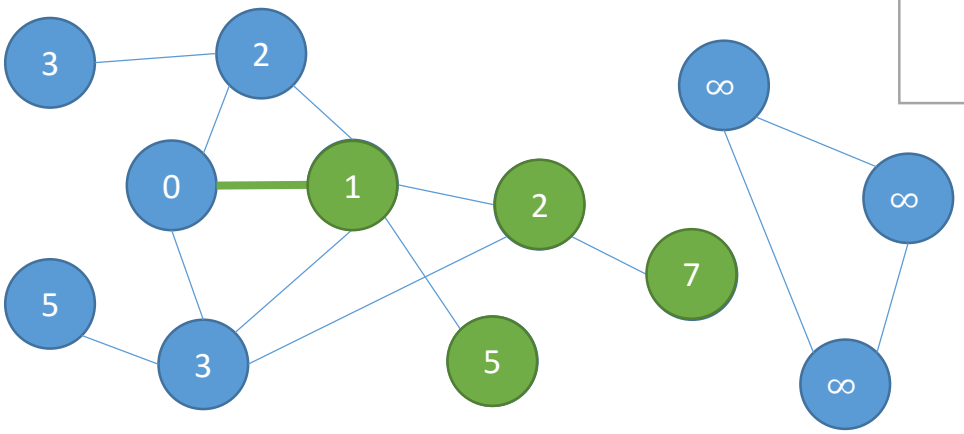
举例：统计图中三角形个数



- 影响增加边所在的两个节点及其邻接点
- 被影响的节点只需计算一次
- 计算结果与各个节点的计算顺序无关

流式图算法特征分析

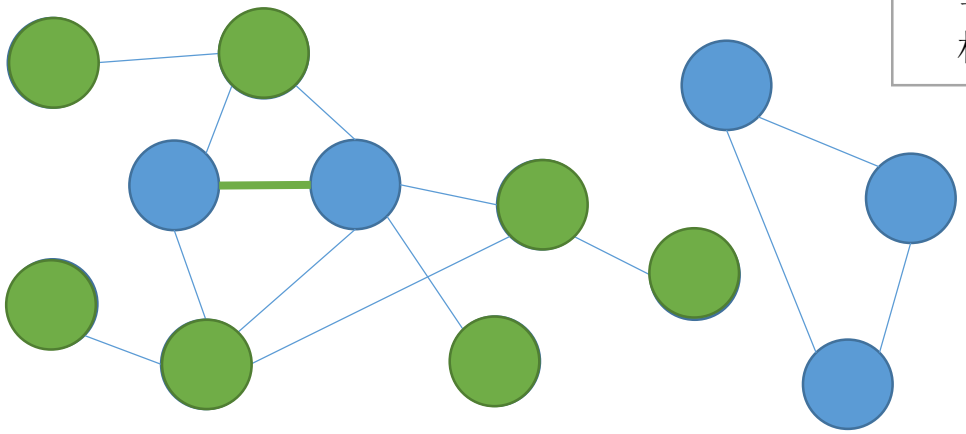
举例：单源点最短路径



- 影响以这两个节点为起点的所有路径
- 被影响的节点只需计算一次
- 计算结果与各个节点的计算顺序无关

流式图算法特征分析

举例：PageRank



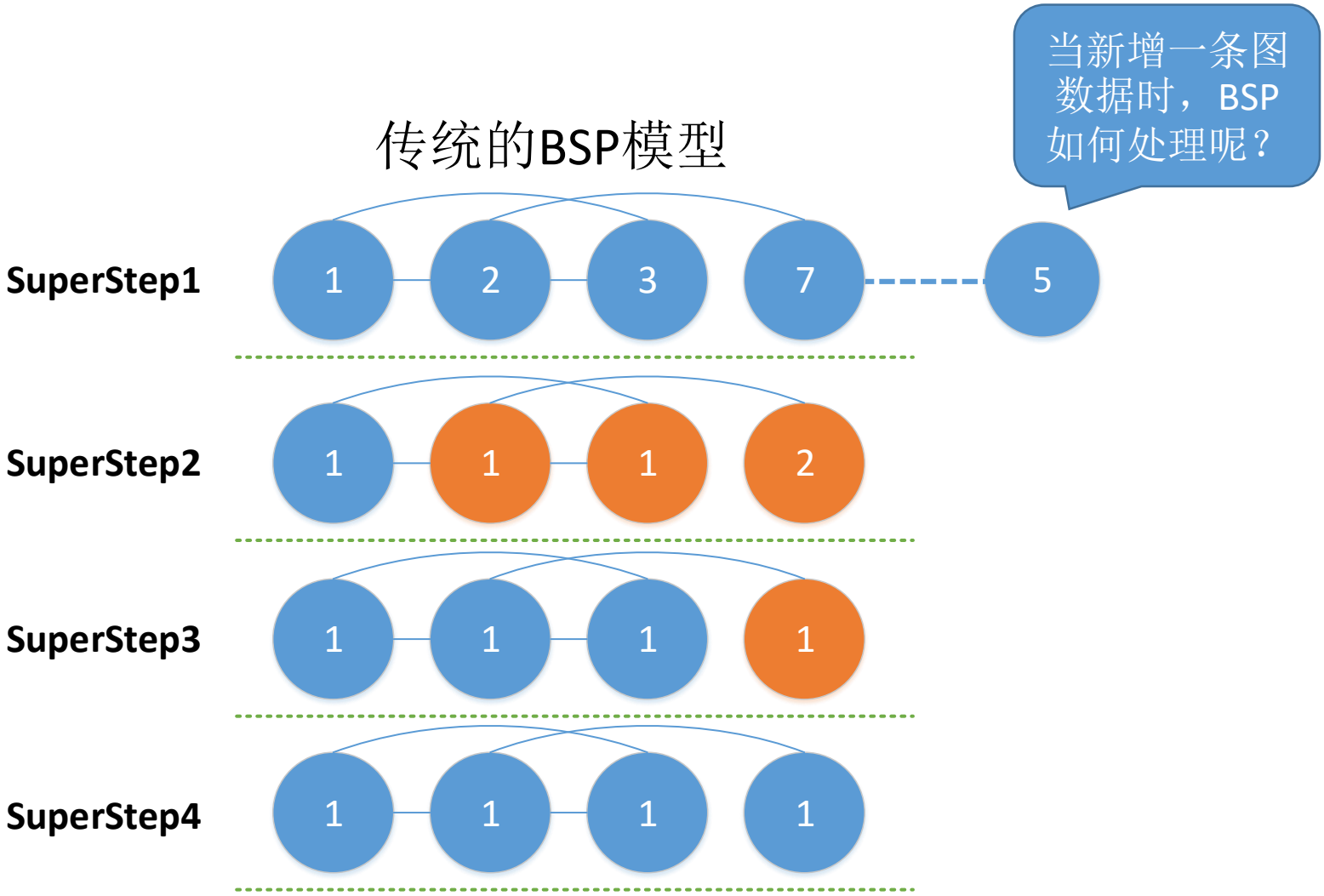
- 影响以这两个节点所在的连通图
- 被影响的节点需要多次迭代计算
- 计算结果与各个节点的计算顺序相关

流式图算法特征分析

增量计算方法　序列一致性　交换律和结合律

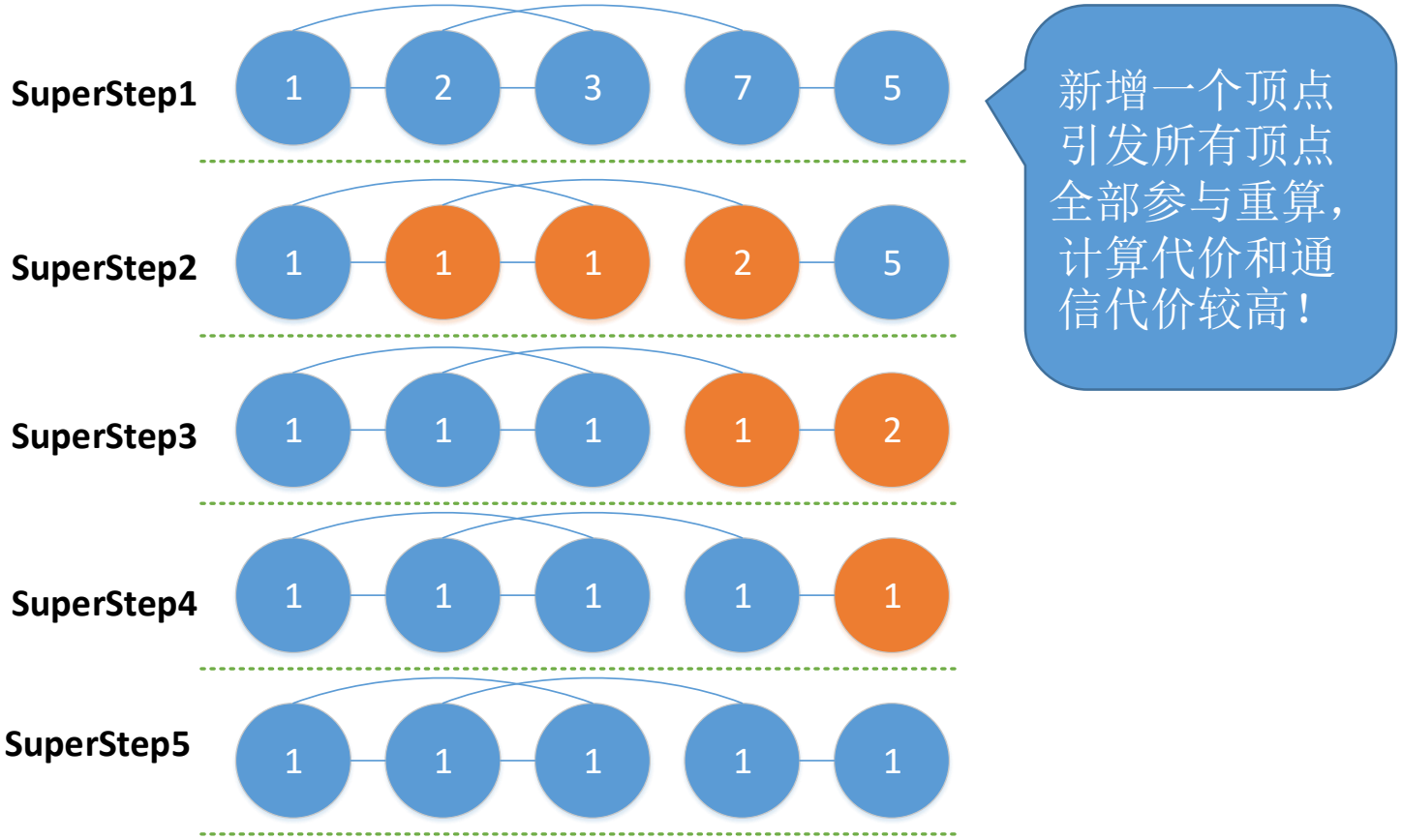
	影响范围	计算方法	计算顺序	计算性质	计算次数
DD	影响新增这条边的源点和目标点	利用原始状态进行增量式计算	最终计算结果和被影响的节点的计算顺序无关	更新函数为加法运算	被影响的节点只参与计算一次
TC	影响新增这条边的源点和目标点，以及这两个点的公共邻接点	利用原始状态进行增量式计算	最终计算结果和被影响的节点的计算顺序无关	更新函数为加法运算	被影响的节点只参与计算一次
SSSP	以这条边的某个节点为起点，沿着某条路径往其他节点传播影响	利用原始状态进行增量式计算	最终计算结果和被影响的节点的计算顺序无关	更新函数为Min运算	被影响的节点可能会参与计算多次
PR	影响这条边的源点和目标点所在的整个连通子图内的所有节点	利用原始状态进行增量式计算	最终计算结果和被影响的节点的计算顺序无关	更新函数为累加运算	被影响的节点一般会参与计算多次

解决方案-模型设计

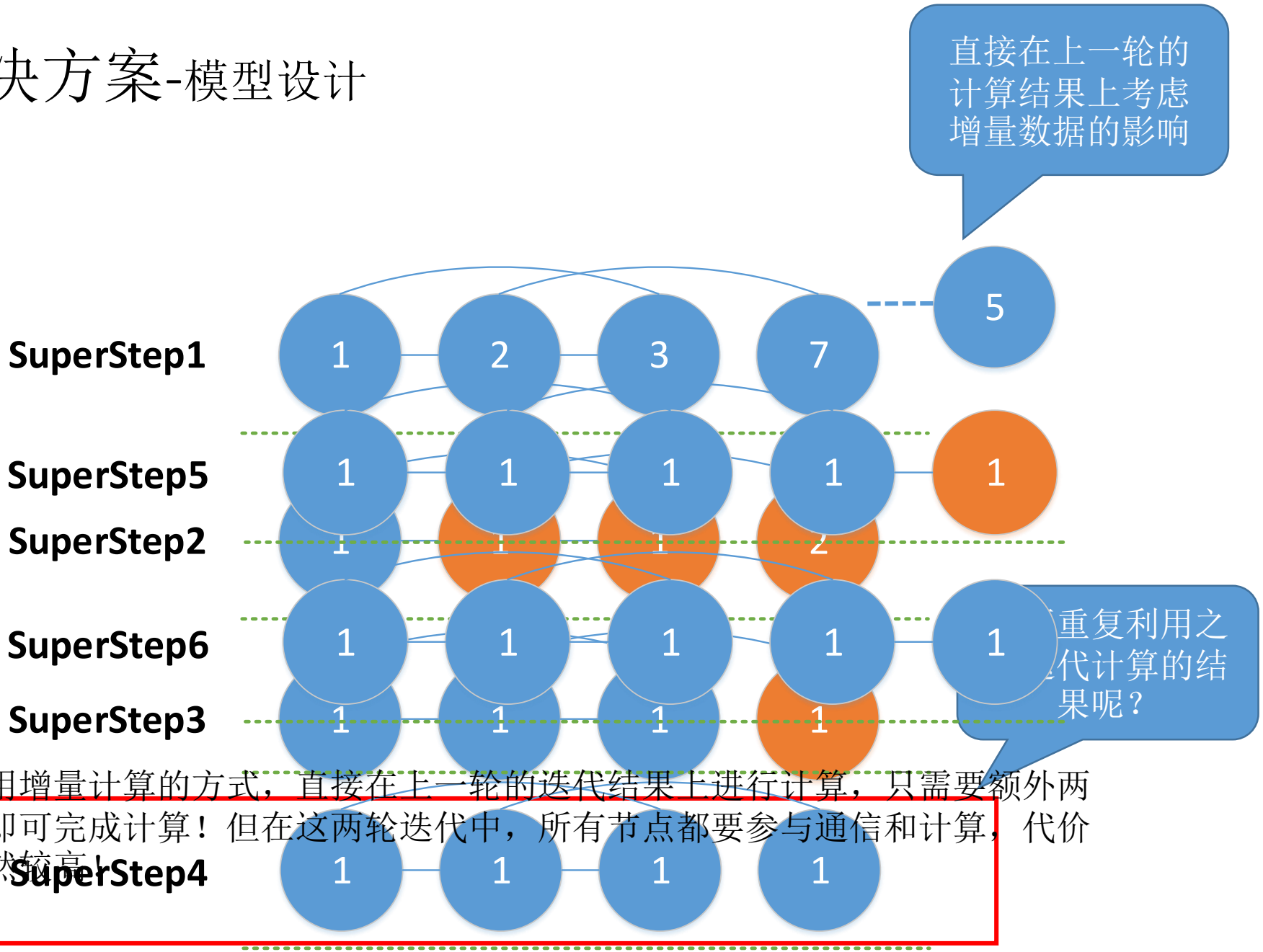

当新增一条图数据时，BSP如何处理呢？
传统的BSP模型
SuperStep1
1
2
3
7
5
SuperStep2
1
1
1
2
SuperStep3
1
1
1
1
SuperStep4
1
1
1
1

解决方案-模型设计

传统的BSP模型针对增量图数据的解决方案

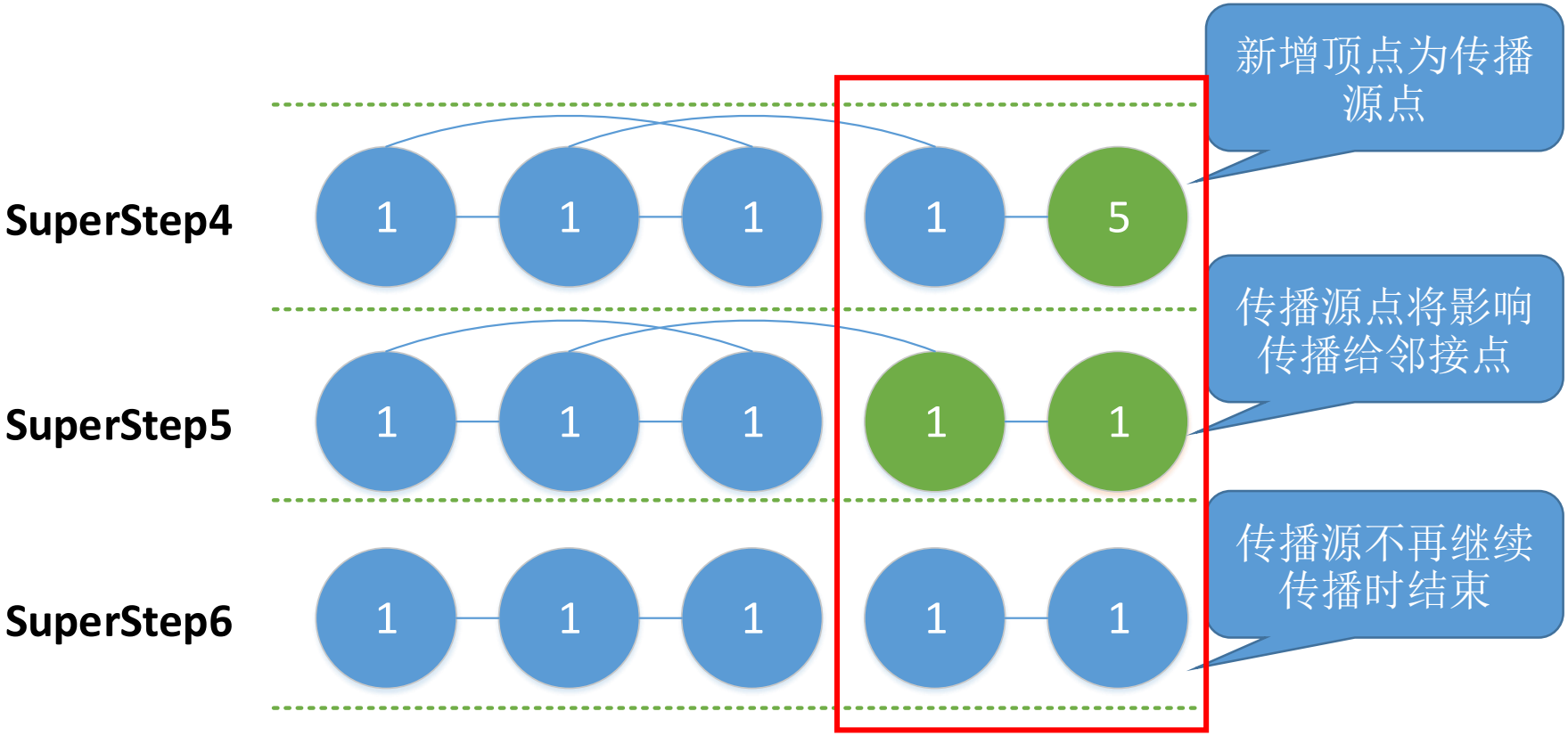


解决方案-模型设计



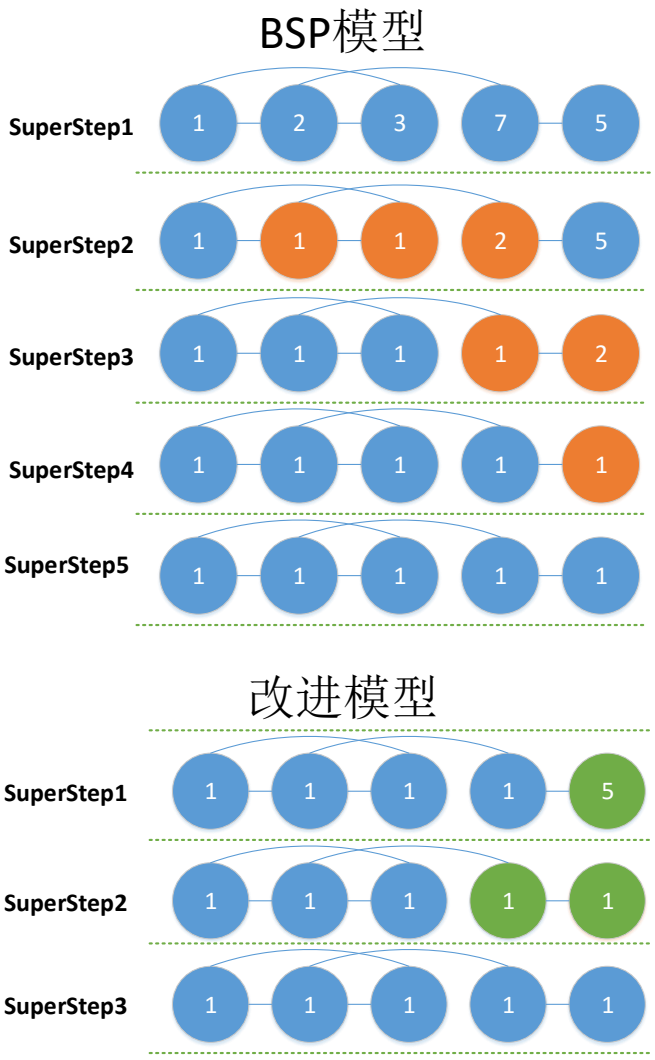
采用增量计算的方式，直接在上一轮的迭代结果上进行计算，只需要额外两步即可完成计算！但在这两轮迭代中，所有节点都要参与通信和计算，代价仍然较高！

解决方案-模型设计



以变化传播的方式对节点进行增量式的更新，有效避免了全图内所有顶点都需参与计算的问题，将影响范围限制在最小域内。

解决方案-模型设计



相比较BSP模型，我们的改进有如下优势：

✓ **增量模型 => 缩短整体迭代所需时间**

✓ **变化传播 => 缩小增量数据影响范围**

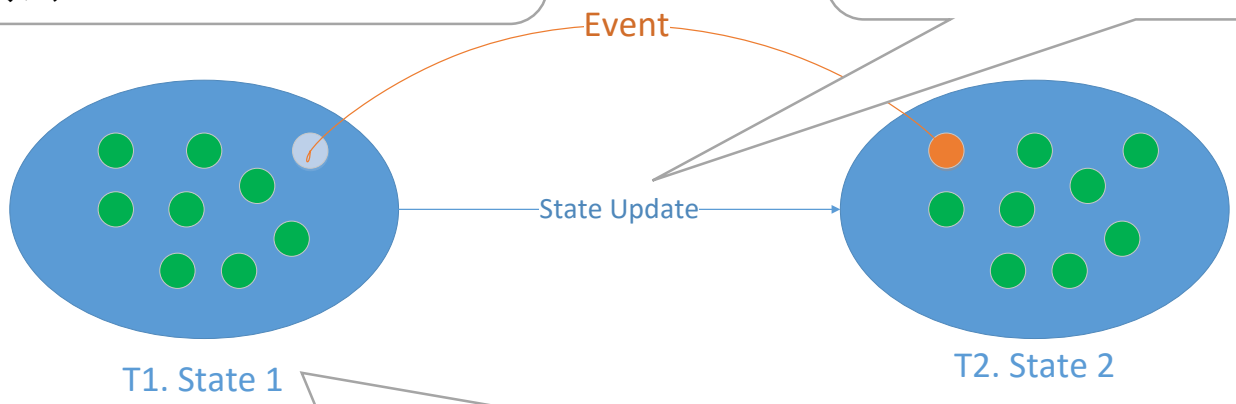
收敛速度更快，参与计算节点更少！

解决方案-模型设计

基于状态更新的图计算模型
采用**增量计算**和**变化传播**的方式更新图的状态

（2）事件（Event）：触发图由T1时刻的State1转换为T2时刻的State2的事件，例如在T2时刻新增加了一条边，将使得图由State1经过某种运算得到State2。

（3）更新（Update）：由事件触发的图的更新过程，即图是如何根据相应的事件来由State1转换成State2。



（1）状态（State）：反应了图当前的特征信息，这些特征信息可以以顶点为单位进行体现，也可以使用用户自定义的特征信息来体现，状态反应了用户的关注点；

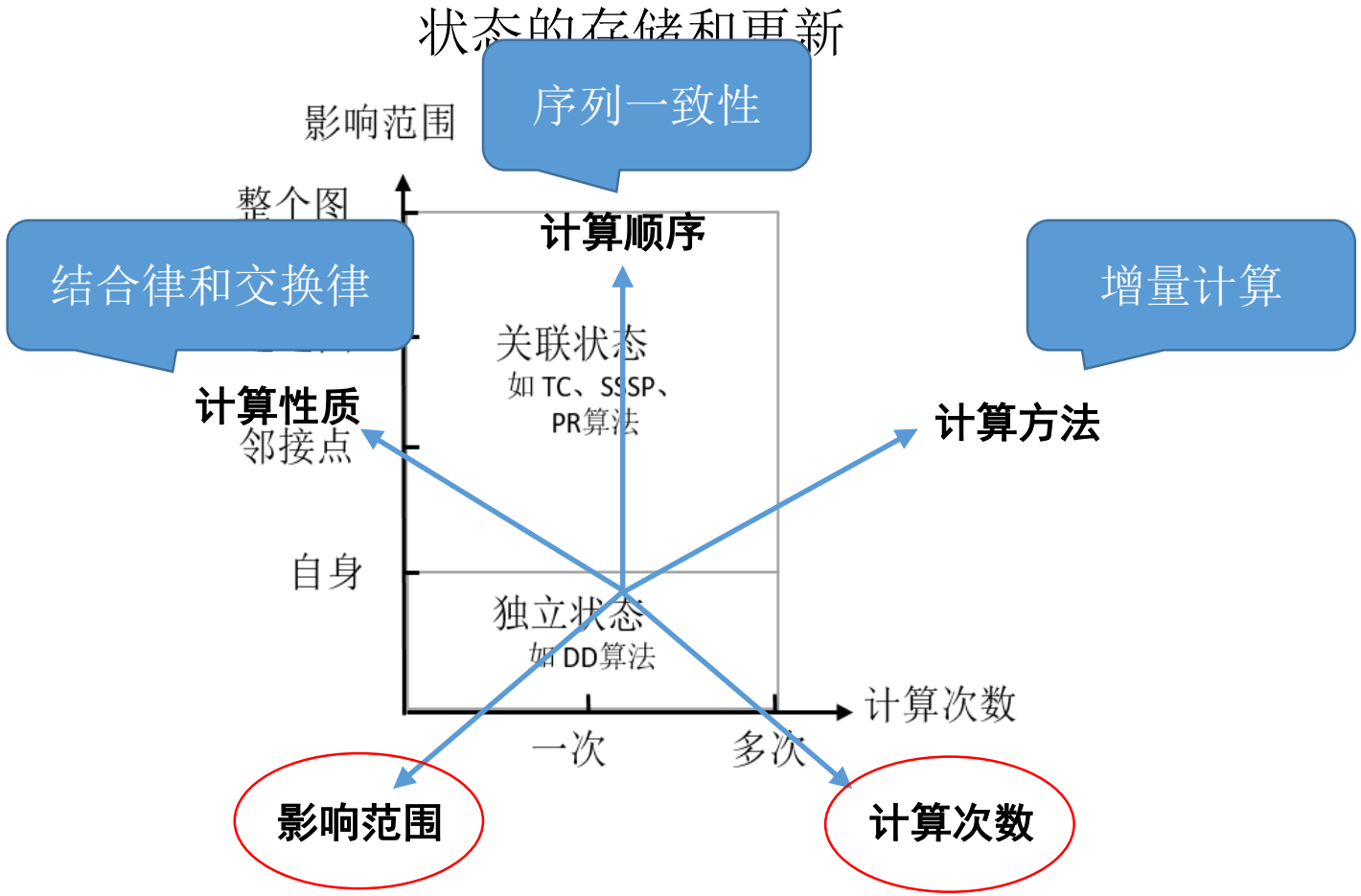
解决方案-模型设计

状态如何存储和更新？

独立状态	关联状态
状态内部的各个**因子**[注1]之间是独立的，一个因子状态的更新不会影响到其他因子状态的跟新。	状态内部的各个因子之间是相关的，一个因子的状态发生变化后，会引起其他因子的状态的更新。
如Degree Distribution算法等。	如PageRank算法， Single Source Shortest Path算法等。

注1：这里所说的因子，是指状态的基本组成单位，如在统计各个顶点的度时，这里的因子就可以是顶点；在统计整个图的顶点数目时，这里的因子就是一个计数器。因此，因子是从用户角度定义的，是状态的基元。

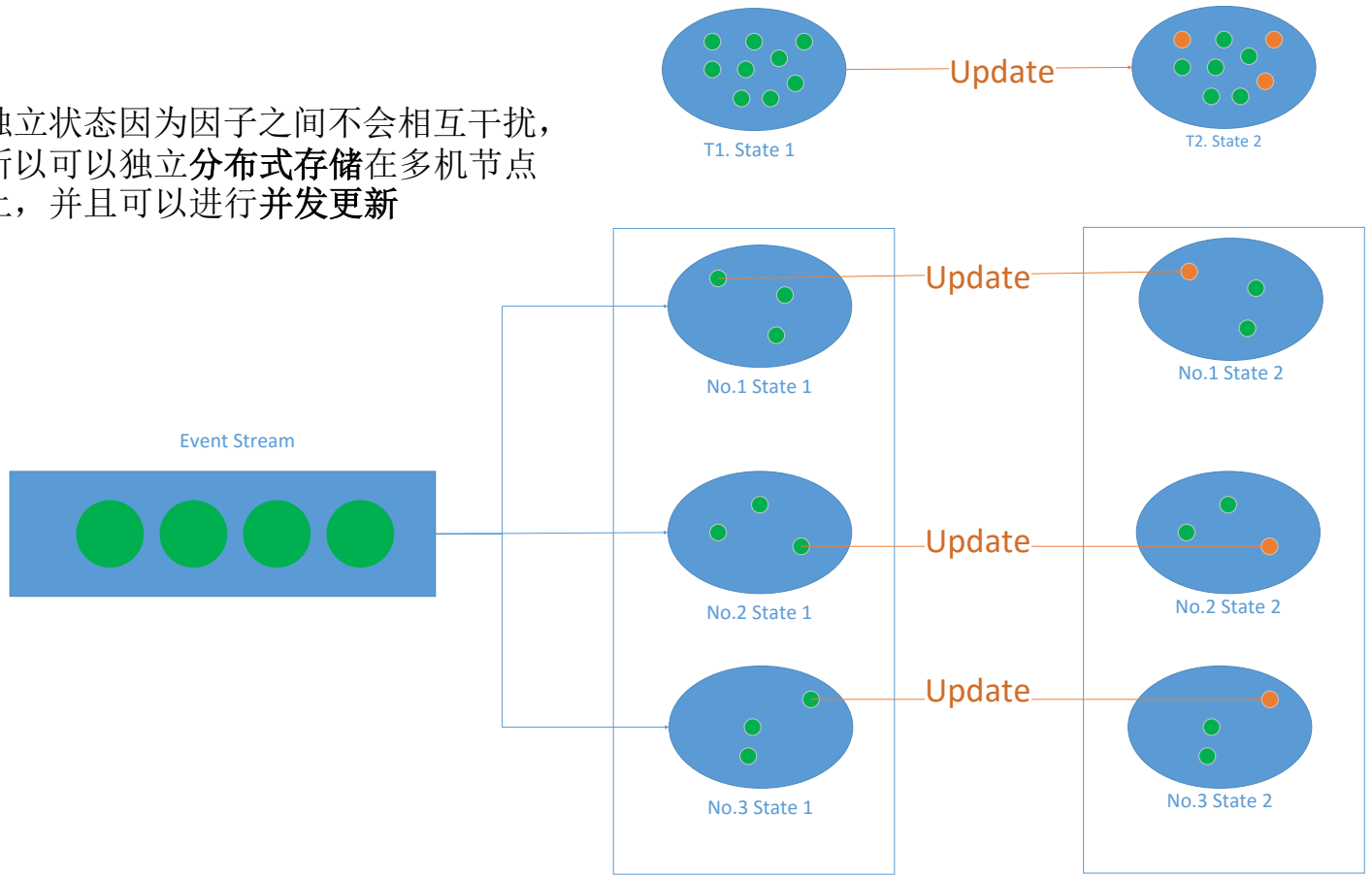
解决方案-模型设计


状态的存储和更新
序列一致性
影响范围
整个图
计算顺序
结合律和交换律
增量计算
关联状态
如 TC、SSSP、
PR算法
计算性质
计算方法
邻接点
自身
独立状态
如 DD算法
计算次数
一次
多次
影响范围
计算次数

解决方案-模型设计

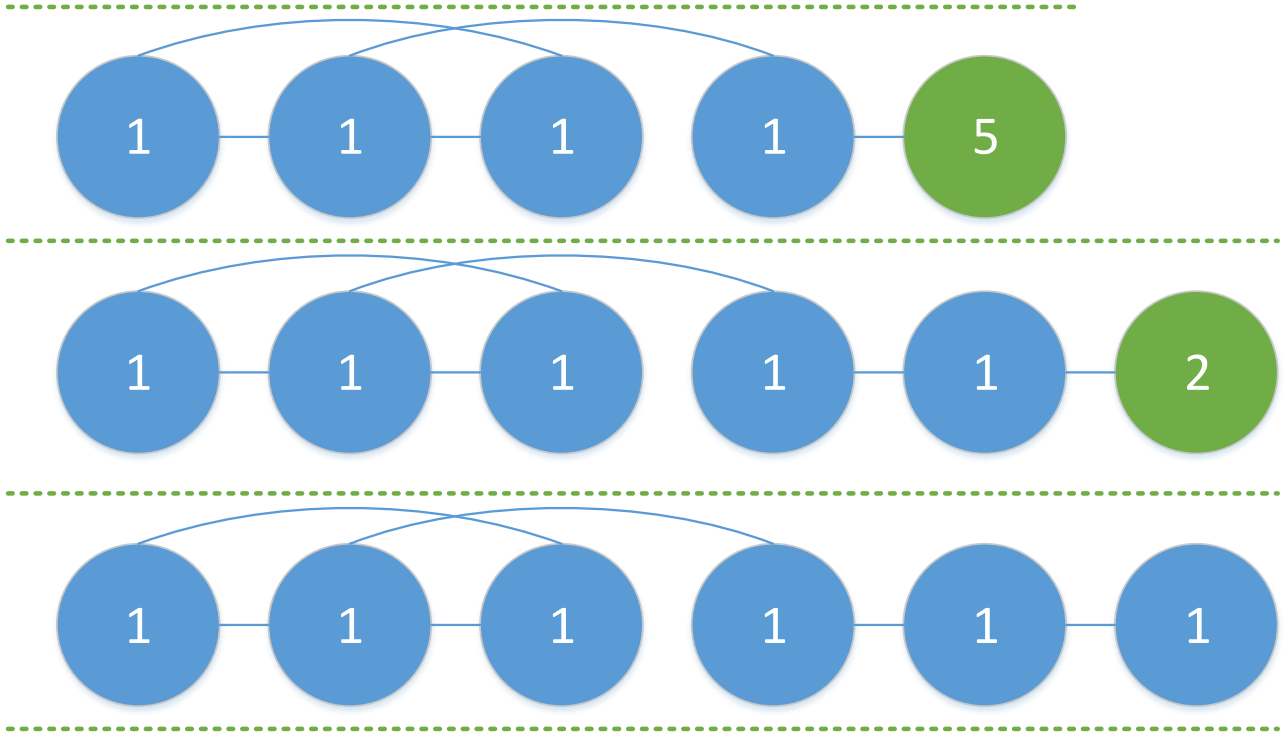
独立状态的存储和更新：分布式存储&并发更新

独立状态因为因子之间不会相互干扰，所以可以独立**分布式存储**在多机节点上，并且可以进行**并发更新**



解决方案-模型设计

关联状态的存储和更新：串行更新

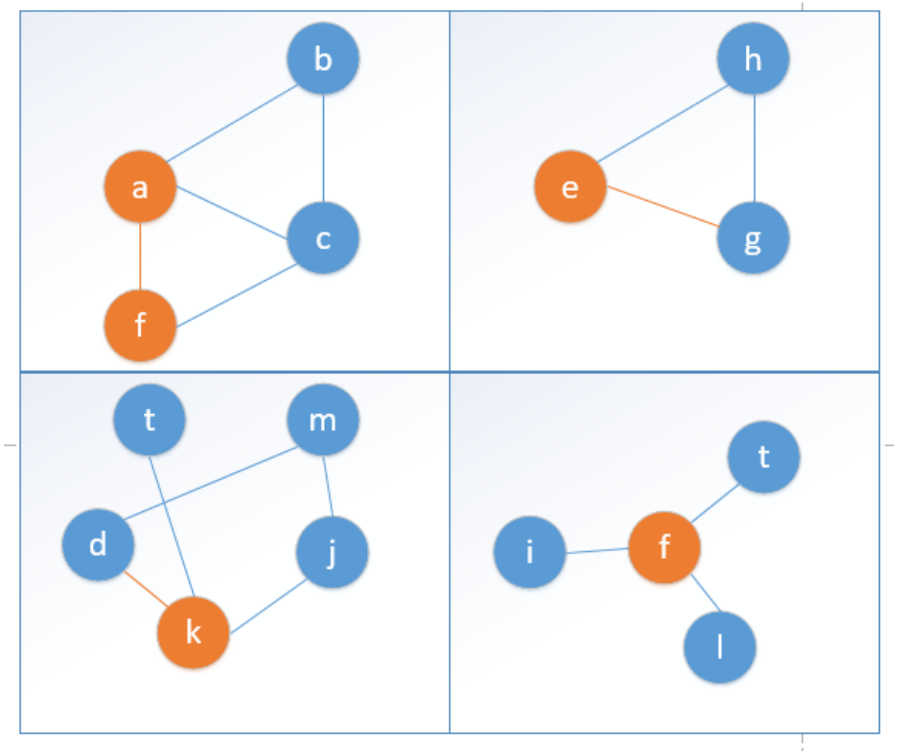


解决方案-模型设计

关联状态的存储和更新：分区并行更新

1. 将原来的图划分成若干个子图，这些子图之间联系是松散的，子图内部联系是紧密的。

2. 子图之间的更新是并行的，子图内部之间的更新是串行的。

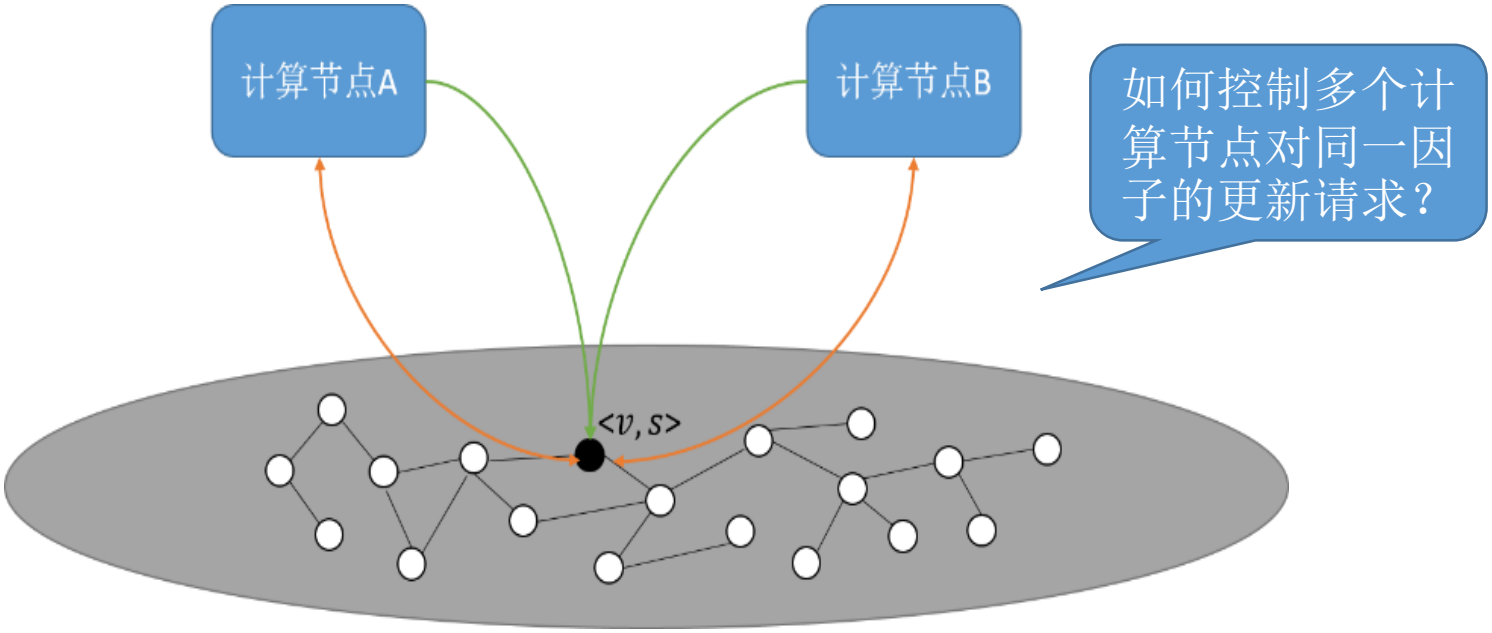


[1] Ioanna Filippidou and Yannis Kotidis. Online and On-demand Partitioning of Streaming Graphs. In 2015 IEEE International Conference on Big Data (Big Data), 2015.

[2] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In KDD '12, pages 1222-1230, 2012.

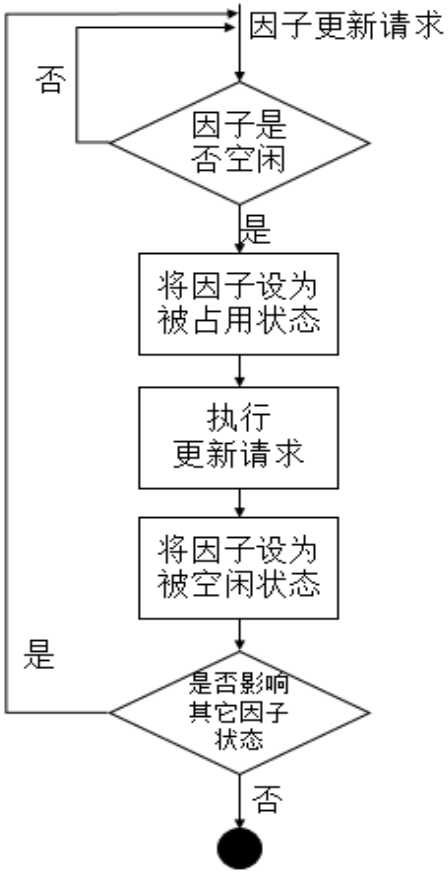
解决方案-模型设计

关联状态的存储和更新：基于细粒度锁更新



解决方案-模型设计

关联状态的存储和更新：基于细粒度锁更新



- 通过加锁的方式，将针对同一个顶点的状态的更新串行化。
- 更新函数满足“增量计算”，“序列一致性”，“交换律和结合律”三个特性。

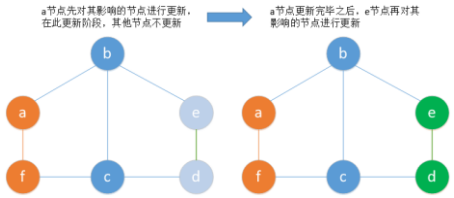
$$state_{new} = f(state_{old}, event)$$

假设有若干个事件$event_1, event_2, \ldots, event_k$，则：

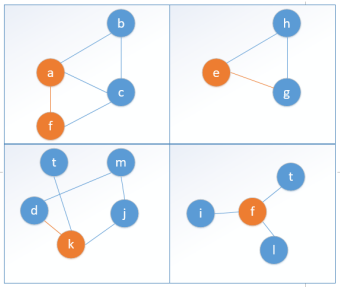
$$state_{new} = f(\ldots f(f(state_{old}, event_{t_1}), event_{t_2}), event_{t_k})$$

解决方案-模型设计

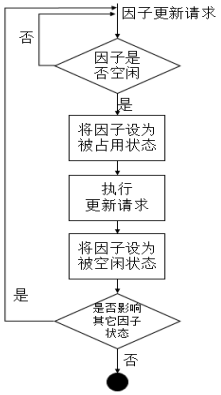
关联状态的存储和更新



串行更新： 实现最简单，也不需要保存中间状态结果，但并行度最低，即使两个不会相互影响的点也不能够同时更新。



分区并行更新：重点是实现合理的分区算法，不需要保存中间状态结果，能够在一定程度上提高并行度，但容易出现倾斜现象。



基于细粒度锁更新：最大程度的提高了系统的并发度，但需要能够灵活控制锁的获取和释放，锁冲突的概率要非常小。

解决方案-架构设计

基于消息传递机制（Pregel）

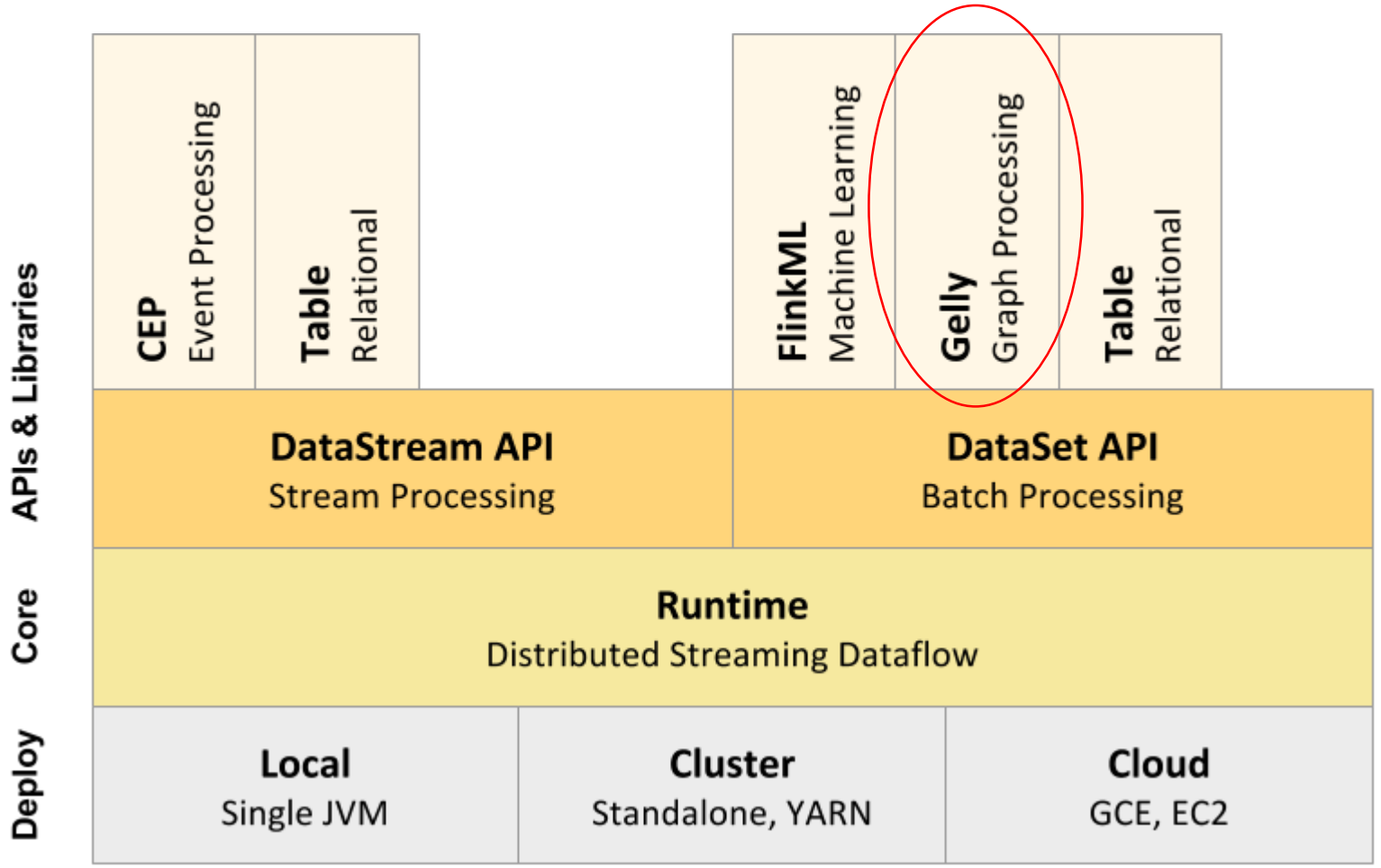
顶点之间通过互发消息交换信息

基于共享内存机制（GraphLab）

顶点通过共享内存直接获取其他顶点或边的状态

解决方案-架构设计

站在巨人的肩膀上：**Flink Gelly**



解决方案-架构设计

站在巨人的肩膀上：Flink Gelly
基于消息传递机制

优势：
- ✓ 丰富的批图计算API
- ✓ 流处理引擎
- ✓ 支持增量迭代计算

不足：
- ✓ 无流式图计算接口
- ✓ 流式迭代接口封装不够完整
- ✓ 没有灵活的分布式数据结构

因为Flink的流式迭代接口不够完整，二次开发的难度较大，因此只能实现简单的诸如TC、DD这样的算法，对于复杂的诸如PR、SSSP算法无法实现，后续将继续改进。

解决方案-架构设计

站在巨人的肩膀上：Hazelcast
基于共享内存机制

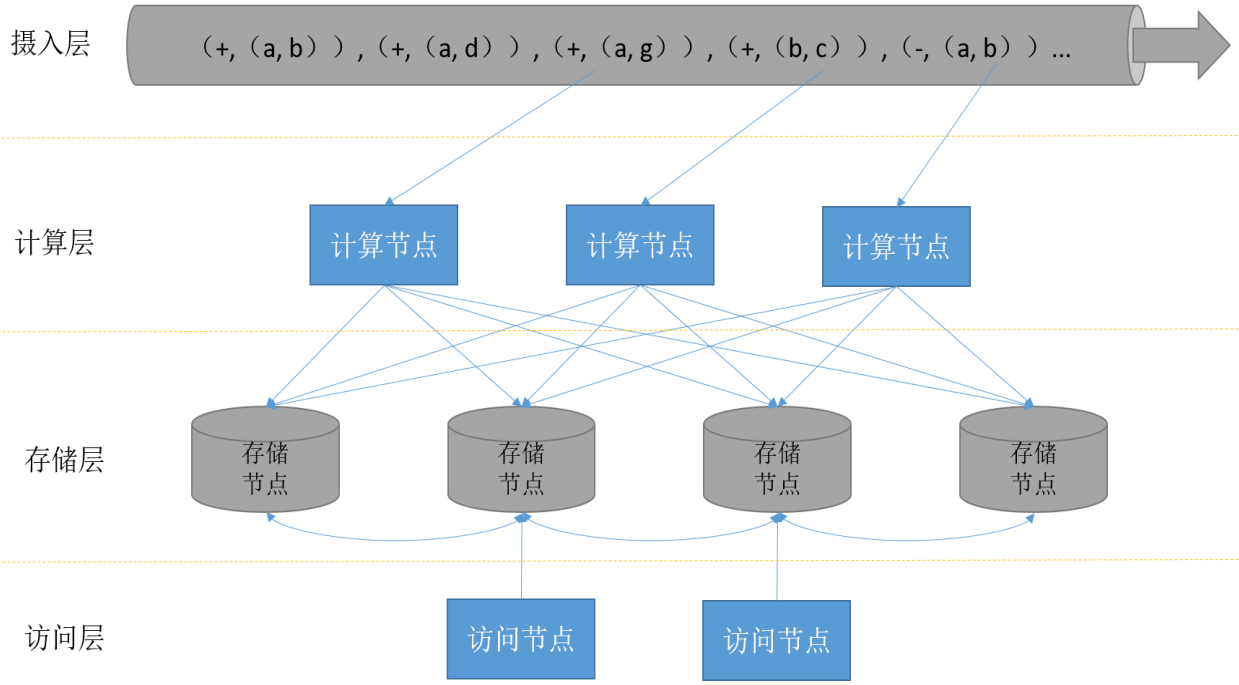
优势：
- ✓ 丰富的分布式数据结构
- ✓ 分布式锁
- ✓ 数据的自动备份和路由

不足：
- ✓ 无图计算接口
- ✓ 流式计算接口不完整

因为Hazelcast提供了丰富的分布式数据结构和分布式锁，可以用来缓存上一次迭代计算的结果，本文基于Hazelcast实现了状态变化传播的流式图计算模型。

解决方案-架构设计

流式图计算系统架构图



- ✓ 计算层：获取存储节点存储的状态，根据用户定义的Transform函数对状态进行更新，并将更新后的状态写回存储层。
- ✓ 存储层：存储所有顶点的状态，方便计算层和访问层的实时访问。

解决方案-算法设计

Triangle Count

Degree Distribution

PageRank

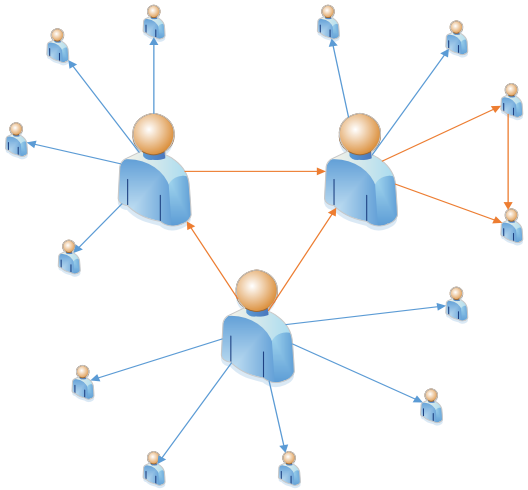
Single Source Shortest Path

……

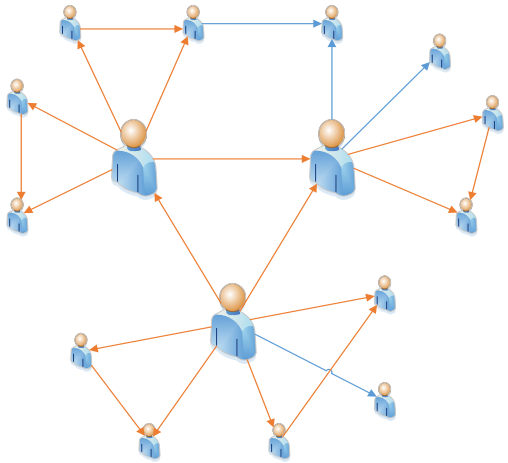
解决方案-算法设计

Triangle Count

Triangle Count算法是用来统计有向/无向图中的不同三角形的数目。该算法在复杂网络分析、链接标签和推荐等多个领域中都是非常基础重要的度量，也是一些诸如复杂网络、聚集系数等图运算中的基本方法。



微博粉丝网络



微信朋友圈网络

解决方案-算法设计

Triangle Count

（1）$State$:图的$State$由每个顶点对应的邻接点的信息组成，即$State = \{s_1, s_2, \dots, s_n\}$，其中$s_k$表示节点$v_k$的邻接点为$N_k$，节点$v_k$构成的三角形的数目为$t_k$，$s_k = (v_k, N_k, t_k)$；

（2）$Event$:图的$Event$为图中新增了一条边，即$Event = \{z_1, z_2, \dots, z_m\}$，其中$z_k$表示新增边$e_k$，$z_k = (e_k, add)$；

（3）Update更新算法：

Algorithm-03 Dynamic Triangle Count

for z_k *in Event*

$N_1 = [], N_2 = []$

$e_k = (v_{v_{k_1}}, v_{v_{k_2}})$

if $s_{v_{k_1}}$ *in State* $N_1 = N_{v_{k_1}} + v_{v_{k_2}}$

else $N_1 = v_{v_{k_2}}$

if $s_{v_{k_2}}$ *in State* $s_{v_{k_2}} = N_{v_{k_2}} + v_{v_{k_1}}$

else $N_2 = v_{v_{k_1}}$

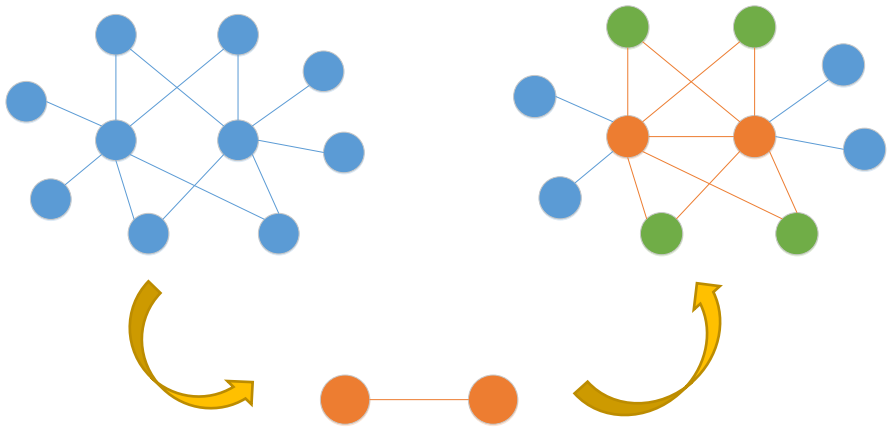
$cross = N_1 \cap N_2$

for c *in cross*

$s_c = (c, N_c, t_c + 1)$

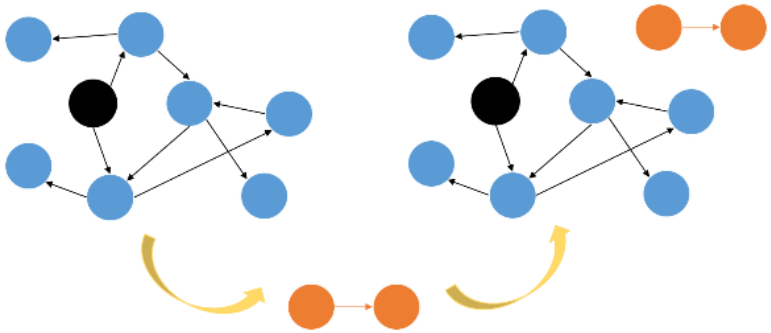
$s_{v_{k_1}} = (v_{v_{k_1}}, N_1, t_{v_{k_1}} + |cross|)$

$s_{v_{k_2}} = (v_{v_{k_2}}, N_2, t_{v_{k_2}} + |cross|)$

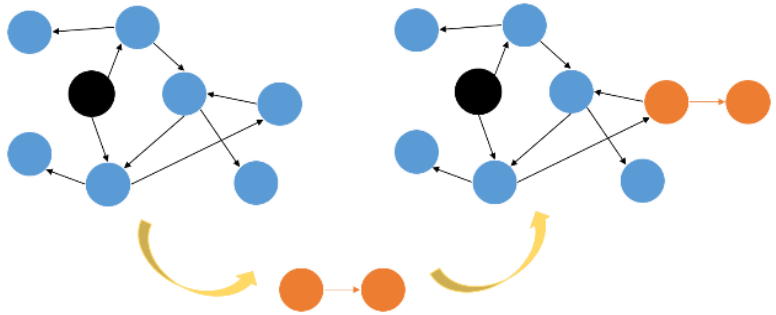


解决方案-算法设计

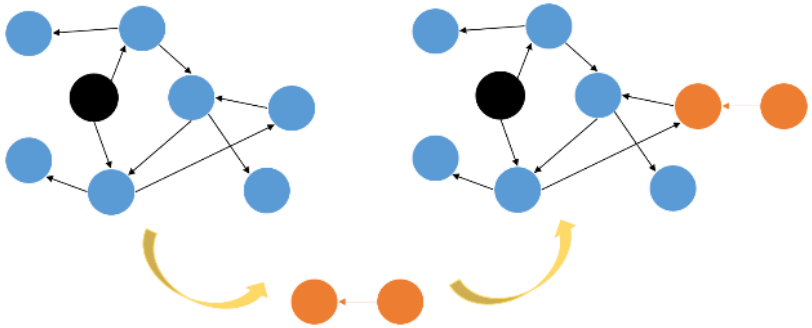
Single Source Shortest Path



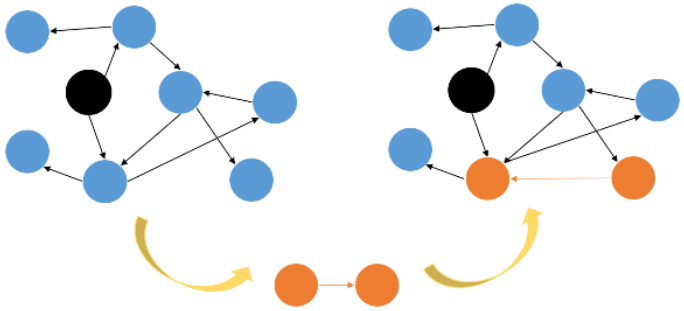
(a) *v1,v2*均为新顶点



(b) v2为新顶点,v1已经存在于系统中



(c) v1为新顶点，v2已经存在于系统中

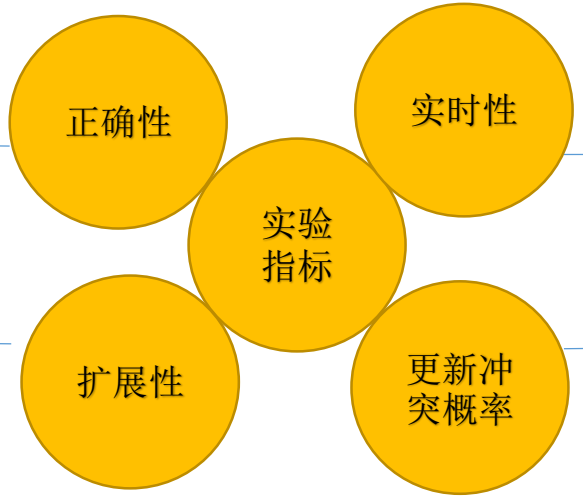


(d) v1,v2均已经存在于系统中

实验结果

系统的算法是正确的，运算的结果正确率在99%以上。

算法能够在执行过程中，实时反馈计算结果，延迟在20ms以内。

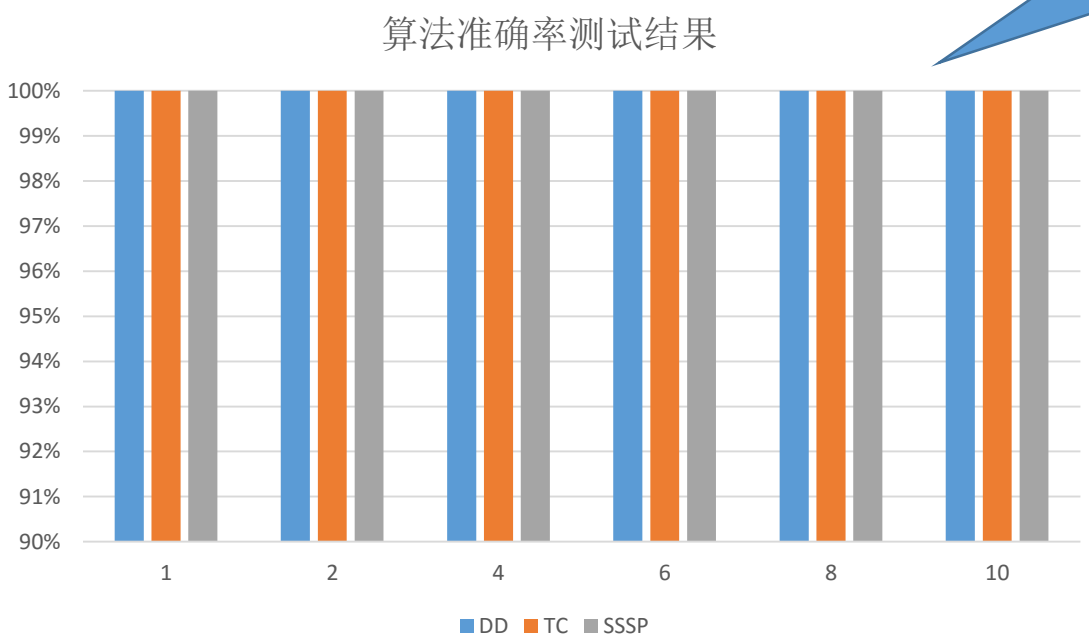


任务能够多机执行，系统具备良好的扩展性。

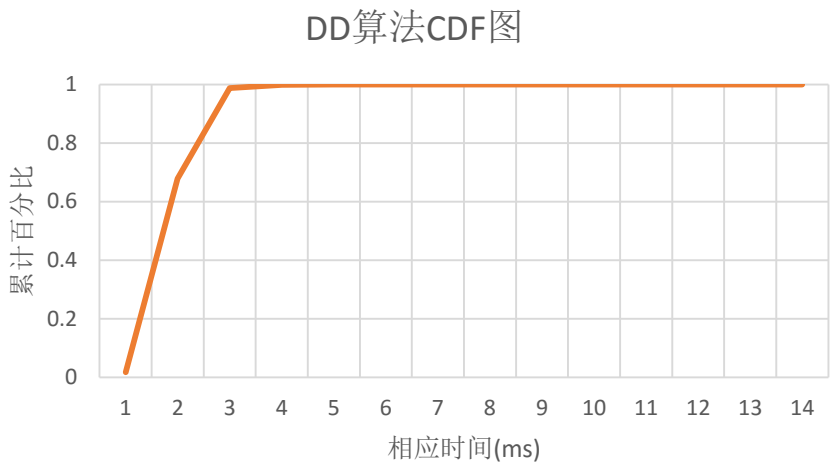
多个计算节点之间发生更新冲突的概率<10%。

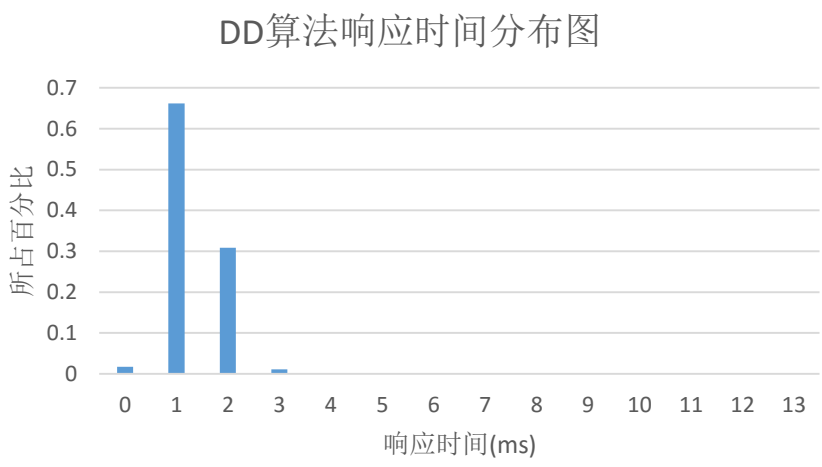
实验结果-正确性

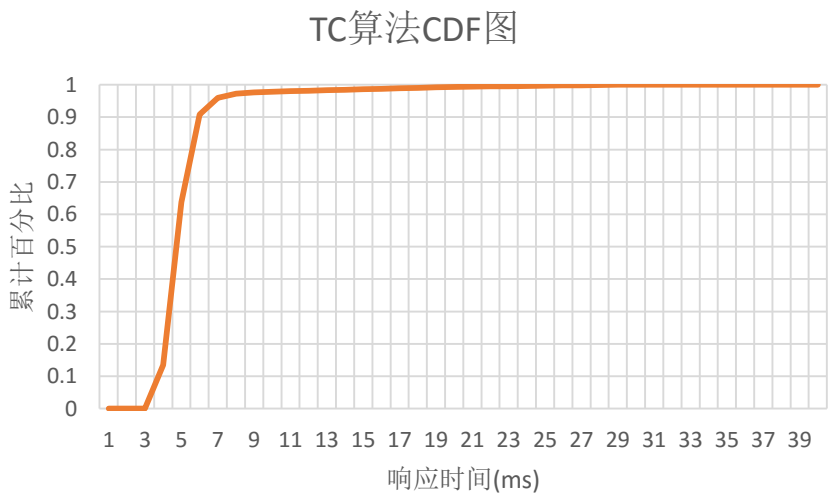
在不同数据规模，不同并发度下算法的正确率仍然保持100%

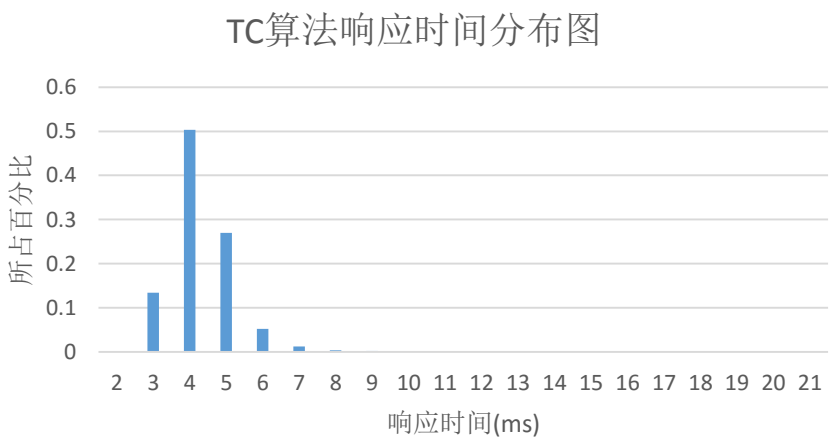


实验结果-实时性


DD算法CDF图
累计百分比
1
0.8
0.6
0.4
0.2
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14
相应时间(ms)


DD算法响应时间分布图
所占百分比
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13
响应时间(ms)

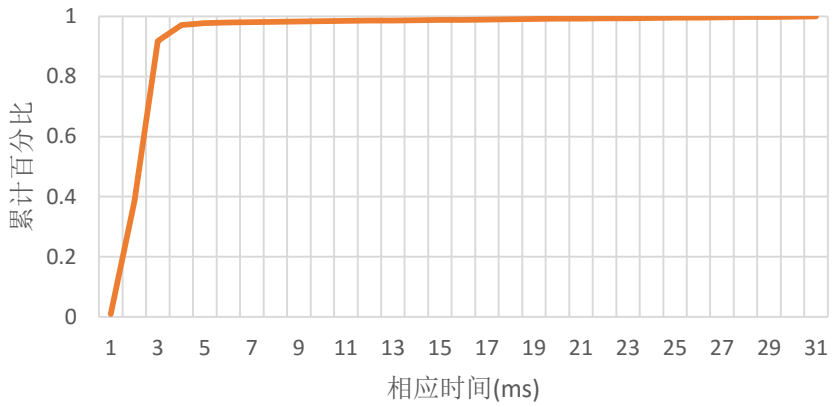

TC算法CDF图
累计百分比
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
响应时间(ms)


TC算法响应时间分布图
所占百分比
0.6
0.5
0.4
0.3
0.2
0.1
0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
响应时间(ms)

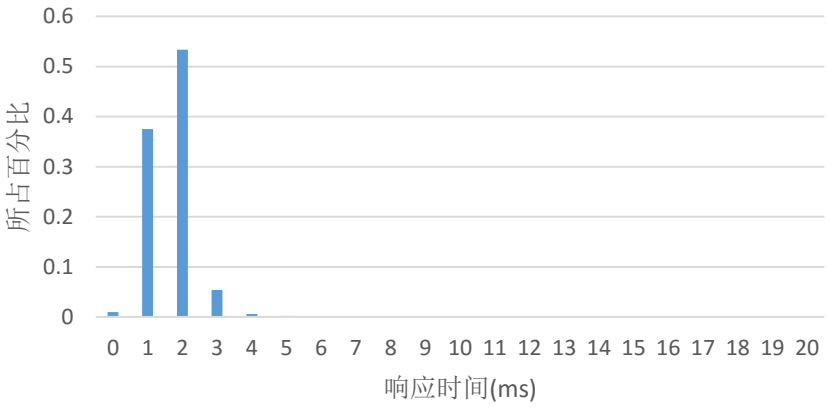
实验结果-实时性

- 从CDF图看，符合长尾效应；
- 从分布图看，90%请求在12ms内得到响应

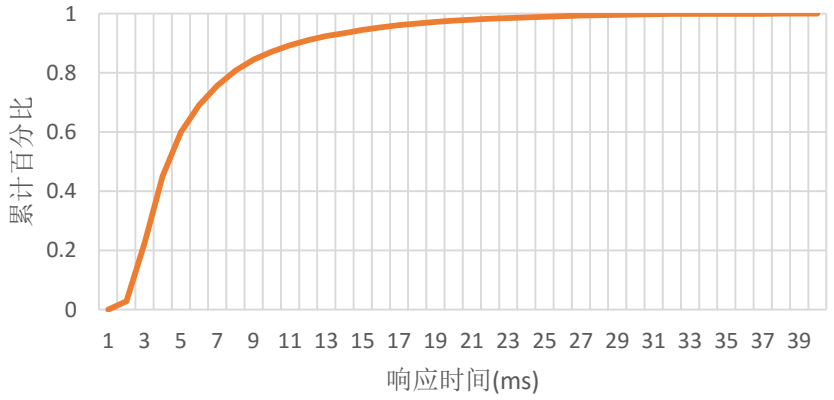
SSSP算法CDF图



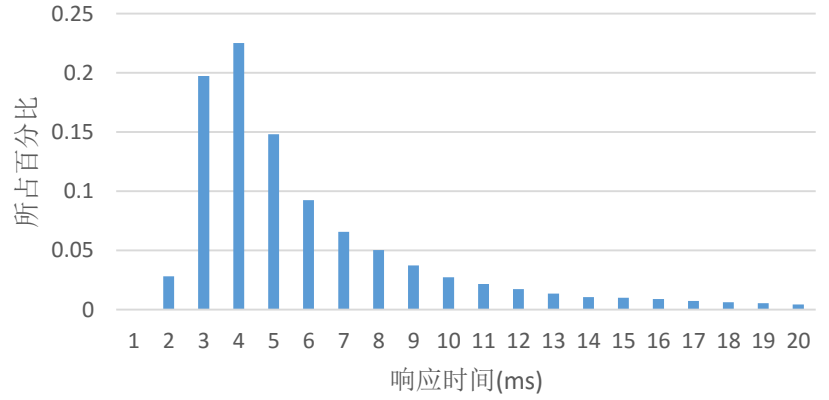
SSSP算法响应时间分布图



PR算法CDF图

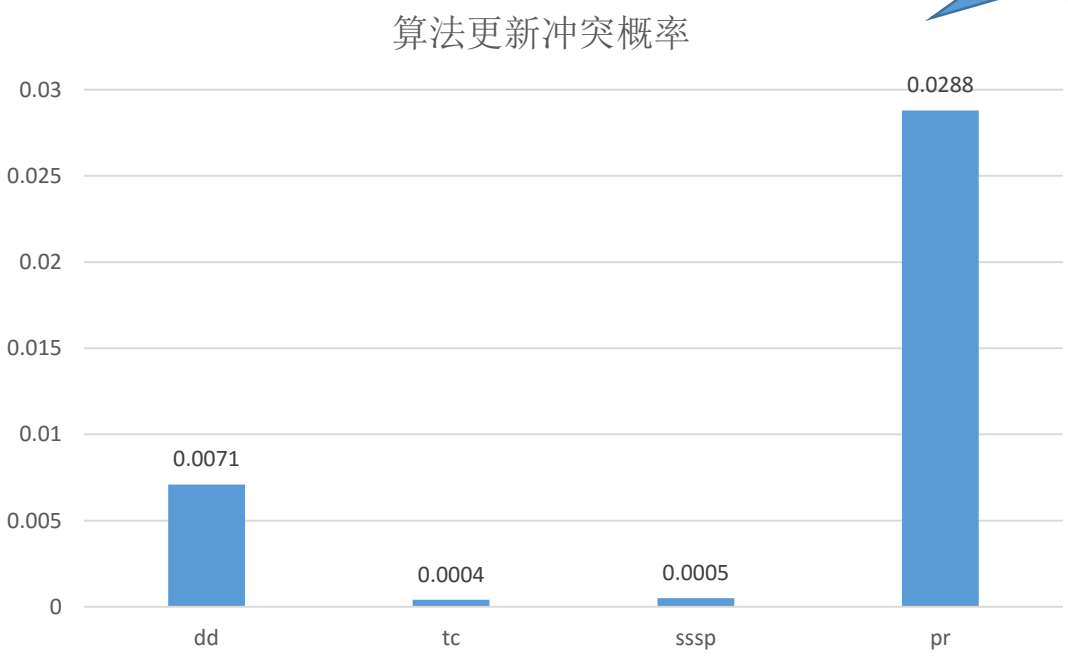


PR算法响应时间分布图

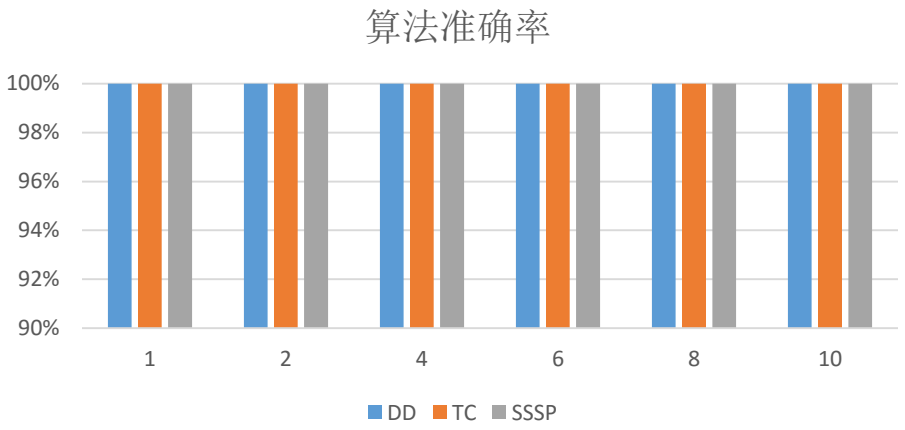


实验结果-更新冲突概率

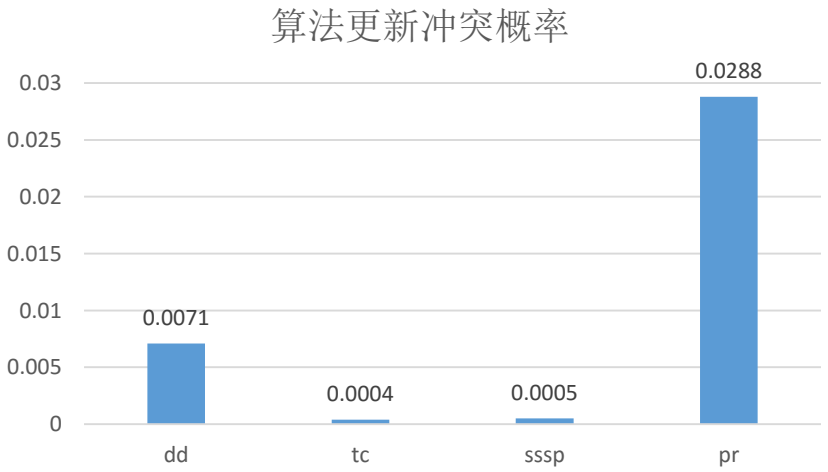
任意两个计算节点的更新冲突概率<3%。



实验结果-扩展性



在1-10个计算节点的扩展中，算法的正确率依旧保持100%



在10个计算节点同时更新过程中，更新冲突概率仍然小于3%

总结和下一步规划

总结：

- ✓ 分析了现有的流式图算法的典型特征。
- ✓ 在满足3个特征的基础上，构建了基于状态更新传播的流式图计算模型。
- ✓ 实现了基于共享内存的流式图计算系统，并完成系统测试。

下一步计划：

- ✓ 系统解决的是增加边的流，能否解决删除边的流，或者增加和删除混合的流？
- ✓ 系统能否应对流速变化的流？吞吐率如何？
- ✓ 本文分析了4个有代表的图算法，对于其他图算法是否适用？
- ✓ 能否在Flink上继续开发，实现基于消息传递的流式图计算系统？