CCTC 2017 中国云计算技术大会
Cloud Computing Technology Conference 2017
CSDN

图计算优化技术探索

章明星，高品，艾智远，**陈康**
清华大学计算机科学与技术系
2017年5月19日

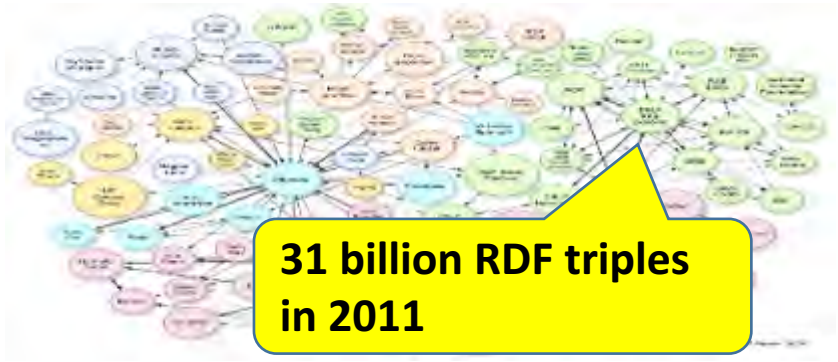
图计算优化技术探索

- 图计算简要介绍
- 使用体系结构局部性加速图计算（IEEE Transactions on Computers）
- 图的三维划分加速计算(USENIX OSDI 2016)
- 外存图计算的加速方法(USENIX ATC 2017)

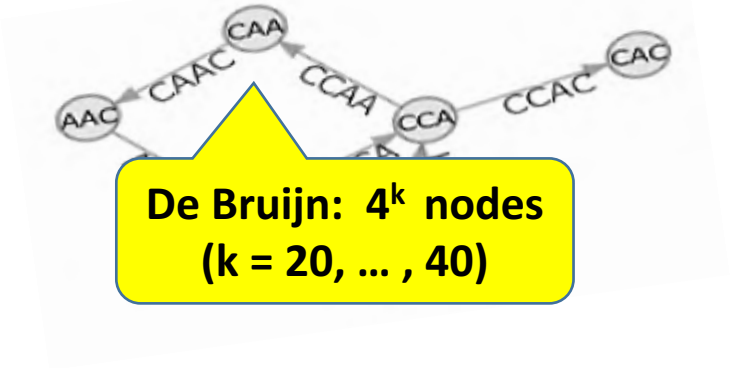
图数据的广泛存在

Web Graph

Social Graph

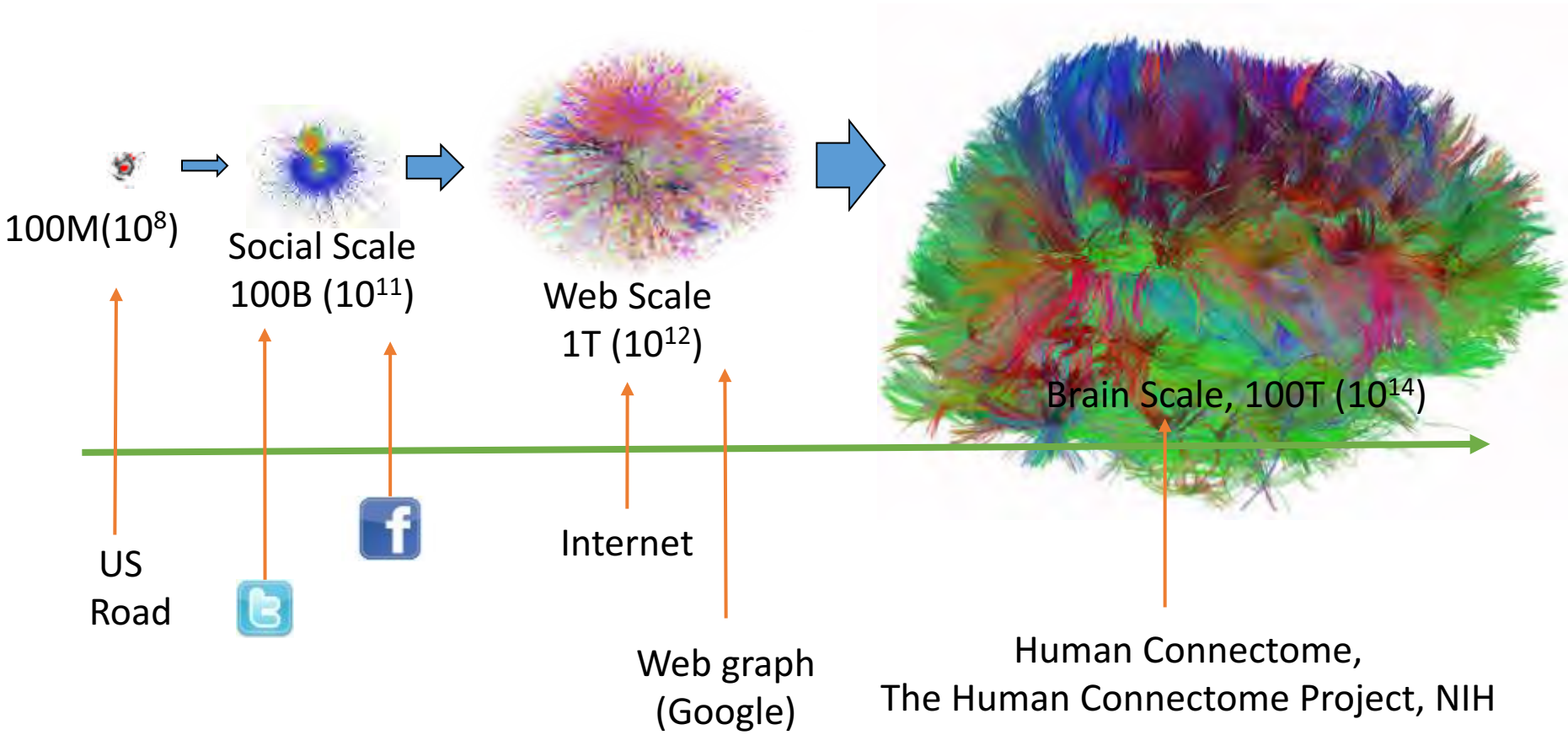


Information Network



Biological Network

图数据规模的迅速增长



Acknowledgement: Y. Wu, WSU

从数据并行到图并行

数据并行 → 图并行

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

Semi-Supervised Learning
Label Propagation
CoEM

Collaborative Filtering
Tensor Factorization

Data-Mining
PageRank
Triangle Counting

图计算的特点

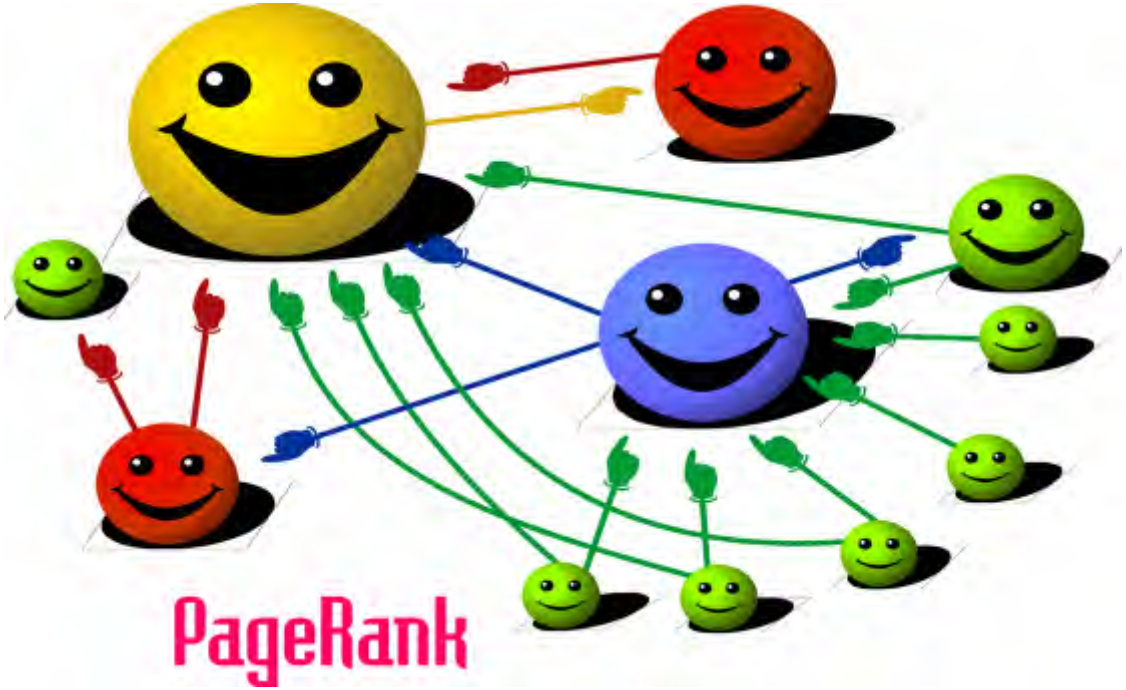
- 四大特点
  - 高访存计算比
  - 数据局部性不好
  - 结构不规则
  - 受数据驱动

图计算的特点

- 四大特点
  - 高访存计算比
  - 数据局部性不好
  - 结构不规则
  - 受数据驱动

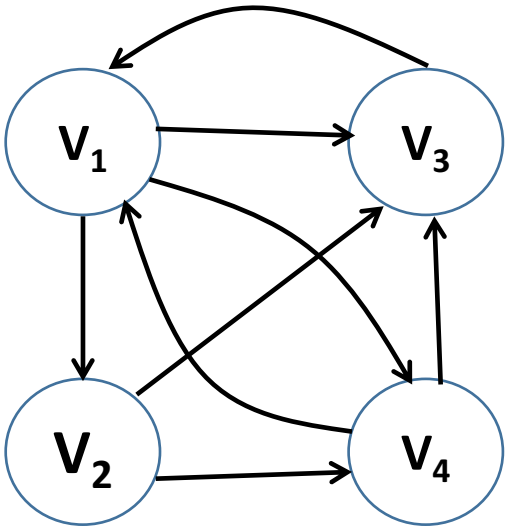
优化数据载入的速度是重中之重

一个典型的图计算PageRank



Acknowledgement: I. Mele, Web Information Retrieval

图的顶点排序：PageRank

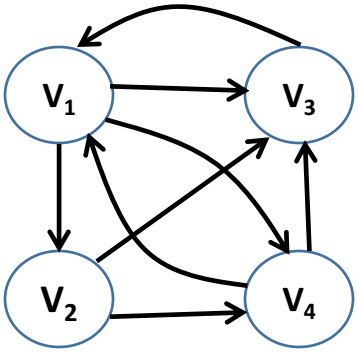


$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

- PR(u): Page Rank of node u
- F_u: Out-neighbors of node u
- B_u: In-neighbors of node u

Sergey Brin, Lawrence Page, "The Anatomy of Large-Scale Hypertextual Web Search Engine", WWW '98

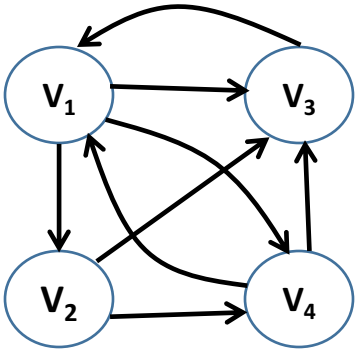
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0
PR(V_1)	0.25
PR(V_2)	0.25
PR(V_3)	0.25
PR(V_4)	0.25

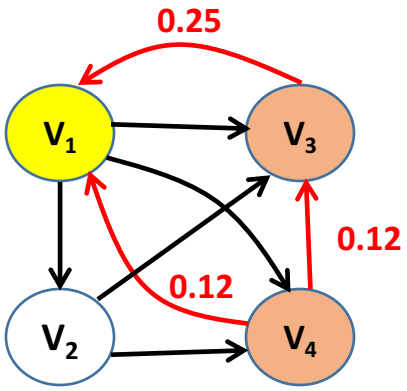
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1
PR(V_1)	0.25	?
PR(V_2)	0.25	
PR(V_3)	0.25	
PR(V_4)	0.25	

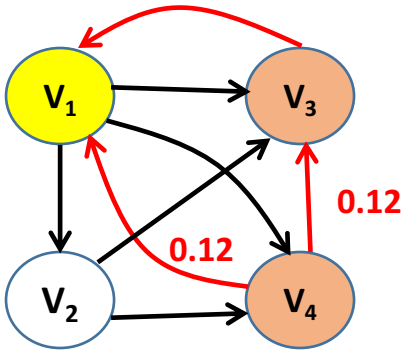
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1
PR(V_1)	0.25	?
PR(V_2)	0.25	
PR(V_3)	0.25	
PR(V_4)	0.25	

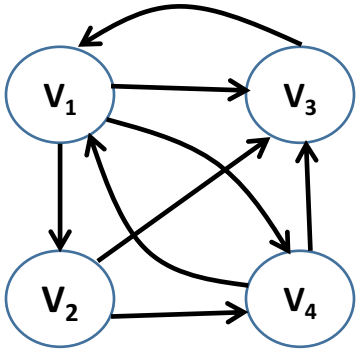
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1
PR(V_1)	0.25	0.37
PR(V_2)	0.25	
PR(V_3)	0.25	
PR(V_4)	0.25	

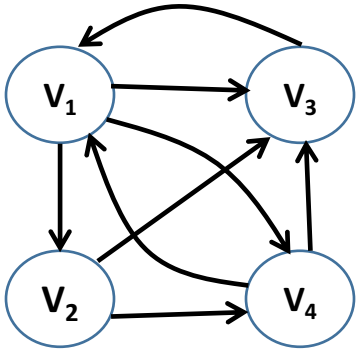
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1
PR(V_1)	0.25	0.37
PR(V_2)	0.25	0.08
PR(V_3)	0.25	0.33
PR(V_4)	0.25	0.20

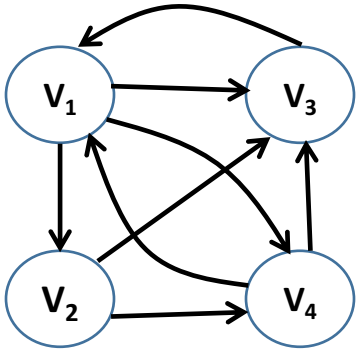
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1	K=2
PR(V_1)	0.25	0.37	0.43
PR(V_2)	0.25	0.08	0.12
PR(V_3)	0.25	0.33	0.27
PR(V_4)	0.25	0.20	0.16

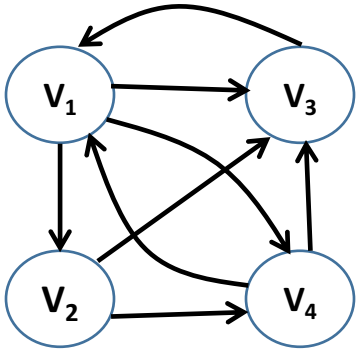
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1	K=2	K=3
PR(V_1)	0.25	0.37	0.43	0.35
PR(V_2)	0.25	0.08	0.12	0.14
PR(V_3)	0.25	0.33	0.27	0.29
PR(V_4)	0.25	0.20	0.16	0.20

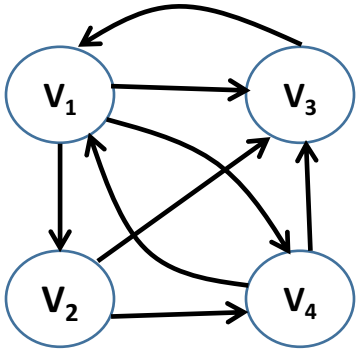
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1	K=2	K=3	K=4
PR(V_1)	0.25	0.37	0.43	0.35	0.39
PR(V_2)	0.25	0.08	0.12	0.14	0.11
PR(V_3)	0.25	0.33	0.27	0.29	0.29
PR(V_4)	0.25	0.20	0.16	0.20	0.19

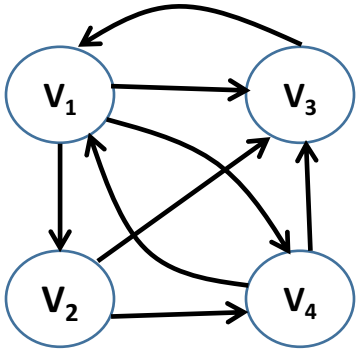
PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

	K=0	K=1	K=2	K=3	K=4	K=5
PR(V_1)	0.25	0.37	0.43	0.35	0.39	0.39
PR(V_2)	0.25	0.08	0.12	0.14	0.11	0.13
PR(V_3)	0.25	0.33	0.27	0.29	0.29	0.28
PR(V_4)	0.25	0.20	0.16	0.20	0.19	0.19

PageRank的迭代计算



$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$

FixPoint

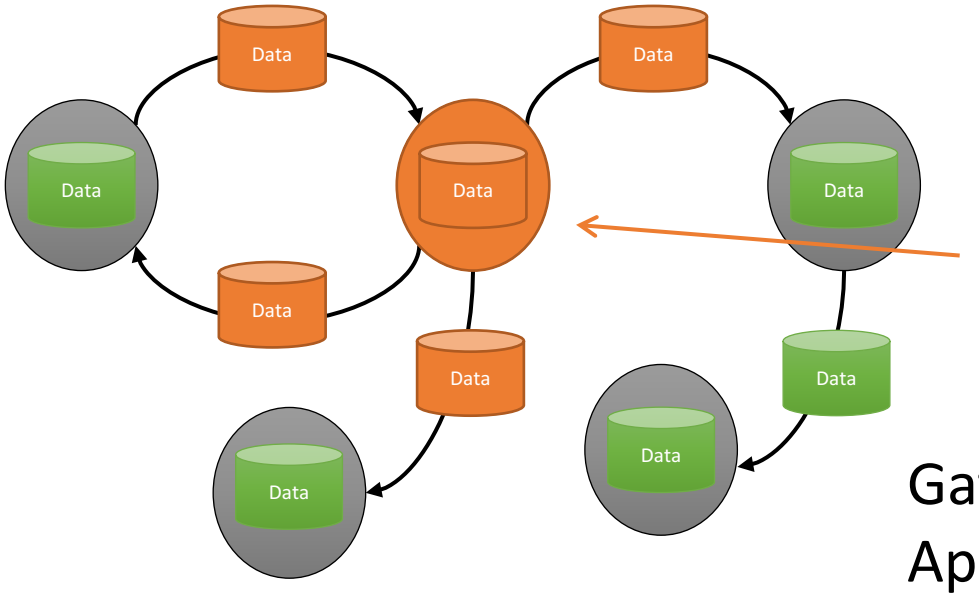
	K=0	K=1	K=2	K=3	K=4	K=5	K=6
PR(V_1)	0.25	0.37	0.43	0.35	0.39	0.39	0.38
PR(V_2)	0.25	0.08	0.12	0.14	0.11	0.13	0.13
PR(V_3)	0.25	0.33	0.27	0.29	0.29	0.28	0.28
PR(V_4)	0.25	0.20	0.16	0.20	0.19	0.19	0.19

图计算系统的计算框架

- 计算框架的作用
  - 便于编程，性能扩展，自动容错
- 以顶点为中心的计算框架
- 以边为中心的计算框架

以顶点为中心的图计算系统

- “Think like a vertex”
- Pregel以及GraphLab开始的编程思想



MyFunc(vertex)
{ // modify neighborhood }

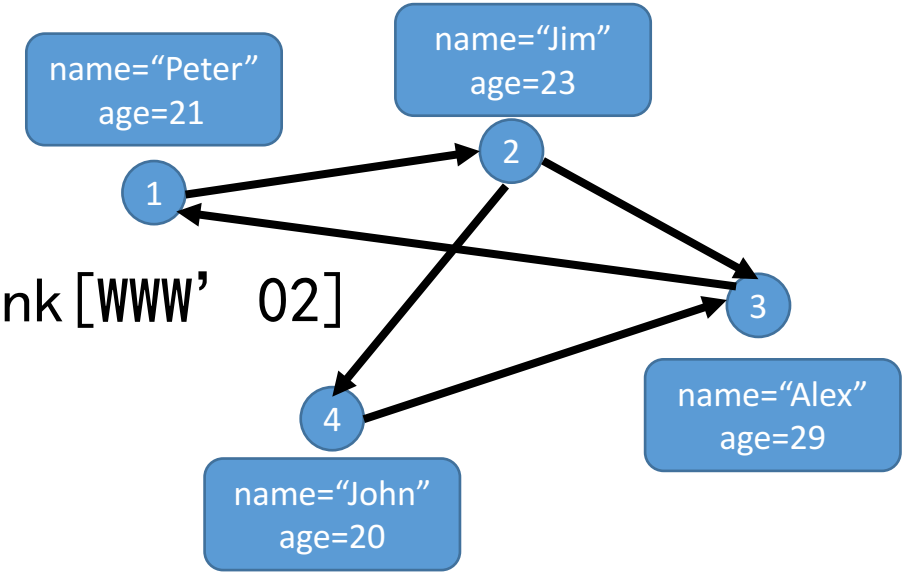
Gather
Apply
Scatter

使用体系结构局部性加速图计算

针对数据局部性的图计算系统设计

- 属性图（Property Graph）
  - 点和边上存在多个属性数据

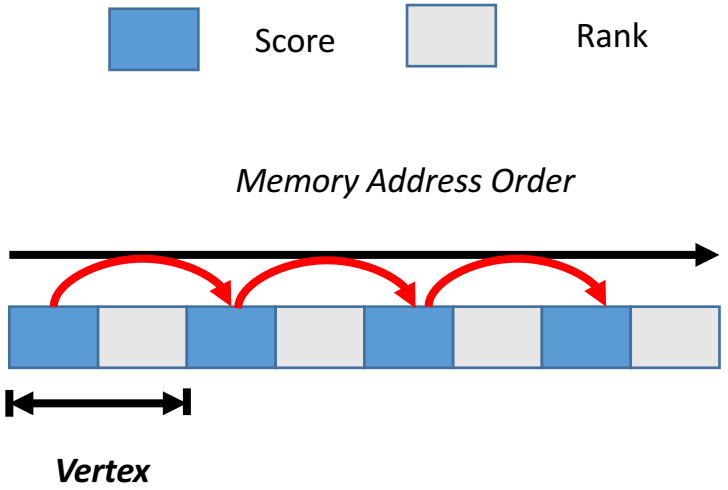
- 有多个属性的图算法
  - Trust Rank[VLDB’04]
  - Topic Sensitive PageRank[WWW’02]
  - SVD++[SIGKDD’08]
  - AdPredictor [ICML’10]



图计算的局部性优化：属性数据局部性

“Think Like a Vertex”

属性数据按照点/边的方式组织在一起（***Vertex View***）



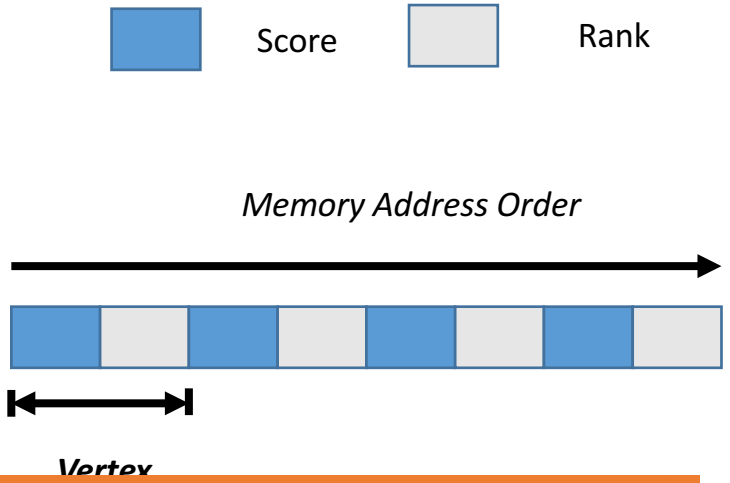
TrustRank算法为例子，在某个计算阶段只访问Score属性。

$$R_j = (1-\alpha)Score_j + \alpha \sum_{(i,j)\in G} R_i \,/\, Degree_i$$

图计算的局部性优化：属性数据局部性

“Think Like a Vertex”

属性数据按照点/边的方式组织在一起（***Vertex View***）



TrustRank算法为例子，在某个计
Score属性

Interleaved Memory Access!!

$$R_j = (1-\alpha)Score_j + \alpha \sum_{(i,j)\in G} R_i \,/\, Degree_i$$

TABLE 1
Effects of Interleaved Memory Access.

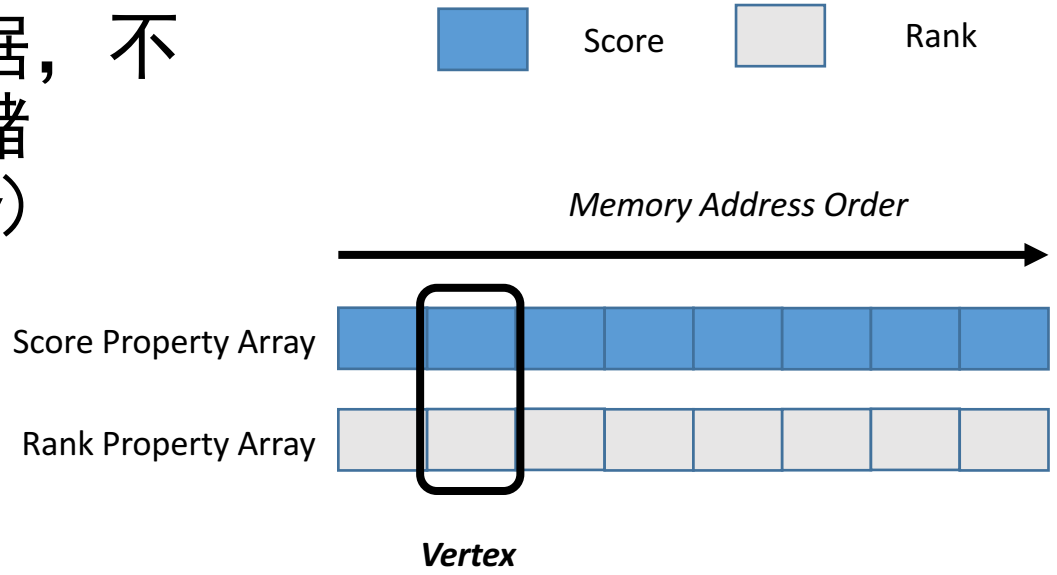
Acc	Conf	BW	DTLB	DCA	DCR	DCW
Suc	8 / 8	7.81	22.65K	238M	266M	189
	64 / 64	7.43	22.12K	199M	214M	191
Int	8 / 16	3.94	38.60K	490M	548M	238
	8 / 32	2.01	78.16K	1.26B	1.29B	276
	8 / 64	1.00	162.06K	2.32B	2.35B	296

DTLB : Data TLB Reads
DCA : Data Cache Access
DCR : Data Cache Read
DCW : Data Cache Write

Note: **x / y** in the *Conf* column means each element size is **y** bytes and only **x** bytes are accessed.

图计算的局部性优化：属性数据局部性

按照属性组织数据，不同的属性分开存储（***Property View***）



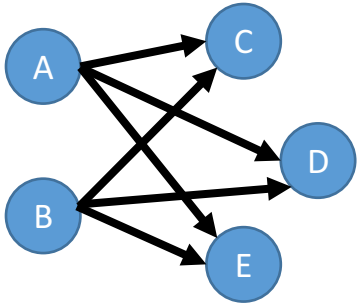
只load需要的数据到cache line，避免内存的间隔访问。

图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）

```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution



对*u*局部性最好，
对*v*局部性最差。

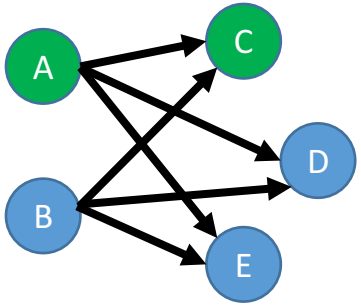
图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）

```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution

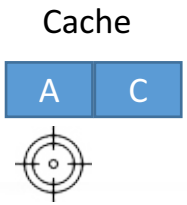
Cache

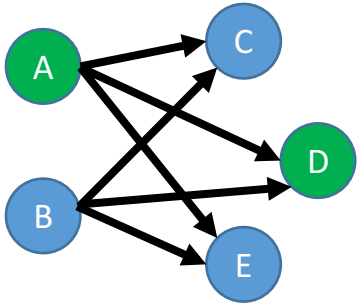


对*u*局部性最好，
对*v*局部性最差。

图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）






```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution

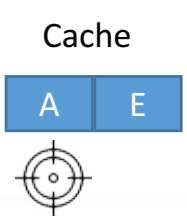
对*u*局部性最好，
对*v*局部性最差。

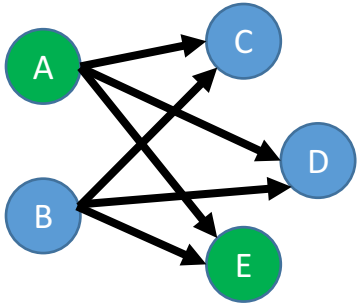
图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）

```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution





对*u*局部性最好，
对*v*局部性最差。

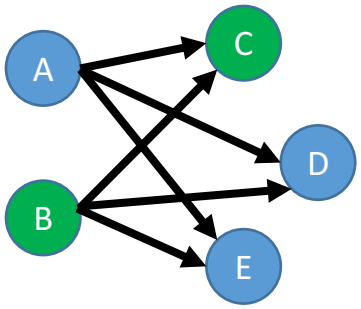
图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）

```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution

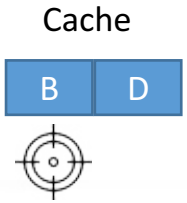
Cache

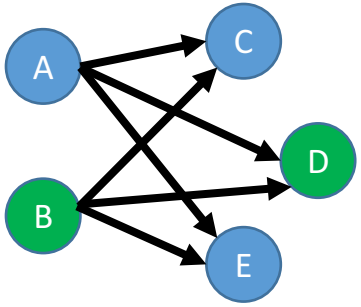


对*u*局部性最好，
对*v*局部性最差。

图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）





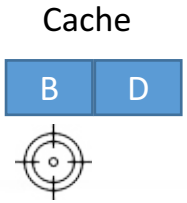

```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

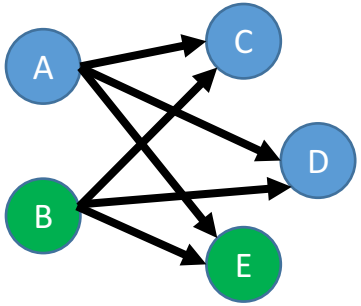
(a) Vertex Centric Execution

对*u*局部性最好，
对*v*局部性最差。

图计算的局部性优化：图结构访问局部性

以点为中心的图结构访问
（Vertex Centric）






```
for u in all vertices :
  for v in neighbors of u :
    compute with vertex[u], vertex[v], edge[u][v]
```

(a) Vertex Centric Execution

对*u*局部性最好，
对*v*局部性最差。

图计算的局部性优化：图结构访问局部性

以边为中心的图结构访问

（Edge Centric）

```
for (u,v) in all edges :
  compute with vertex[u], vertex[v], edge[u][v]
```

(b) Edge Centric Execution

通过对边进行合理的排序，达到对所有点访问的局部性都好。

图计算的局部性优化：图结构访问局部性

```
for (u,v) in all edges :
  compute with vertex[u], vertex[v], edge[u][v]
```

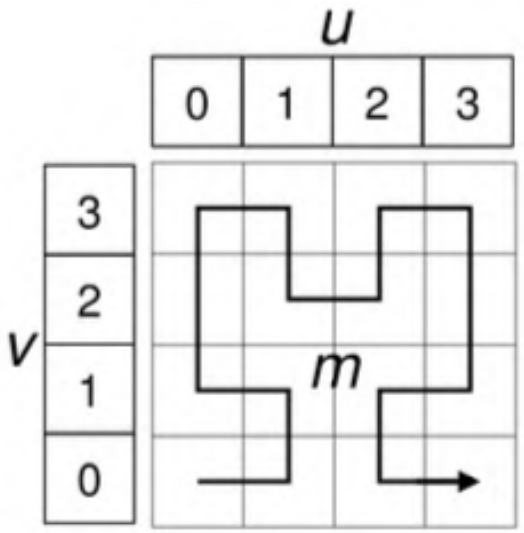
(b) Edge Centric Execution

以边为中心的图结构访问

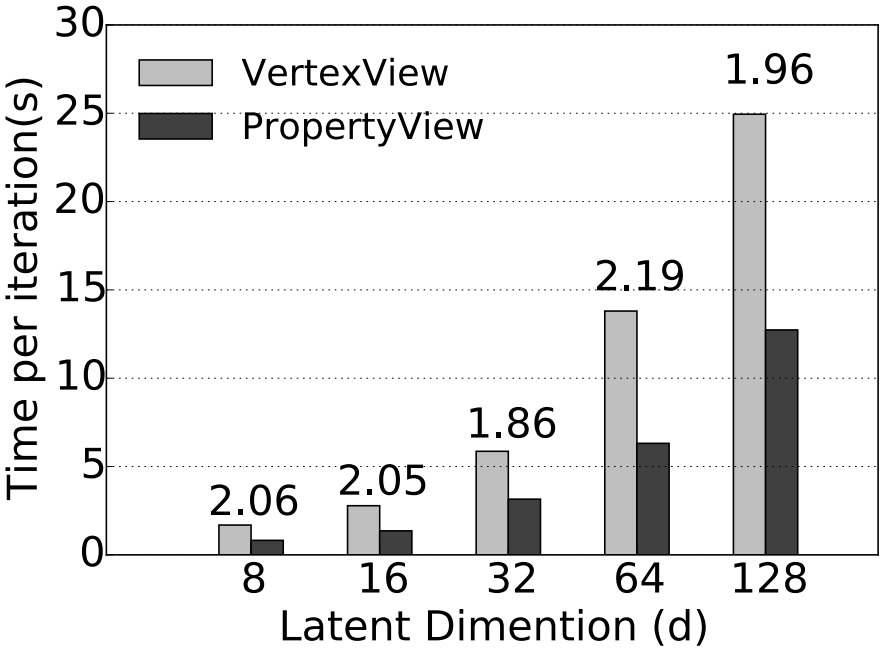
（Edge Centric）

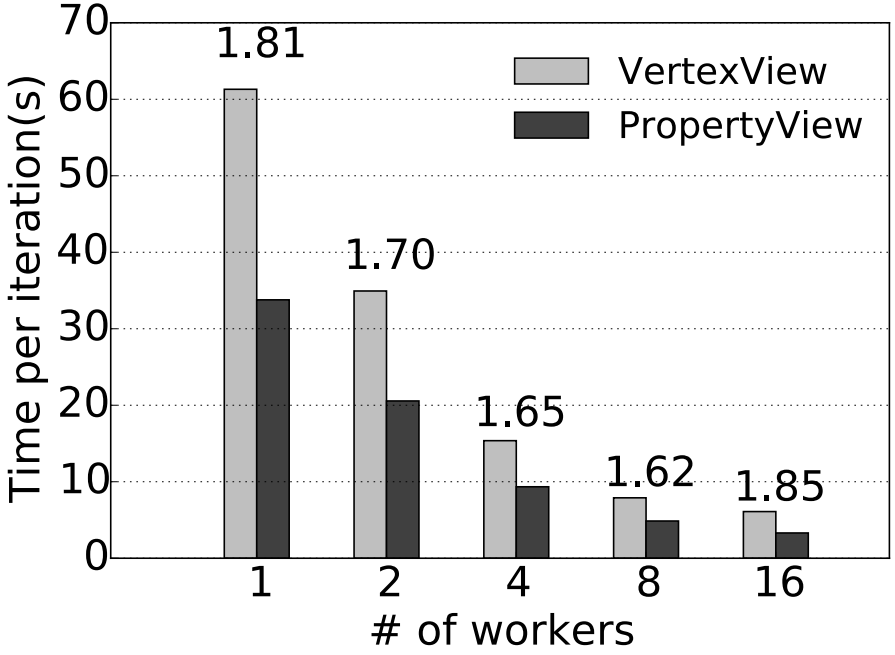
通过对边进行合理的排序，达到对所有点访问的局部性都好。

实现时采用Hilbert Order。

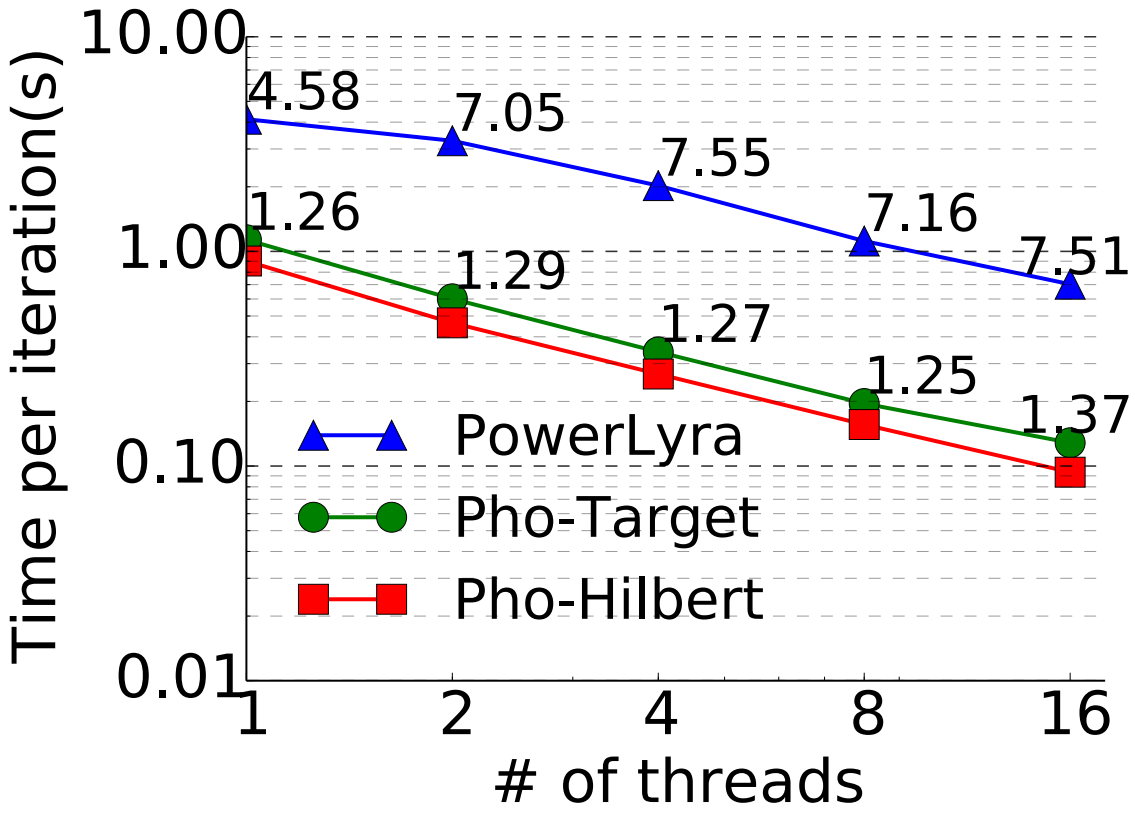


性能提升（Property View）

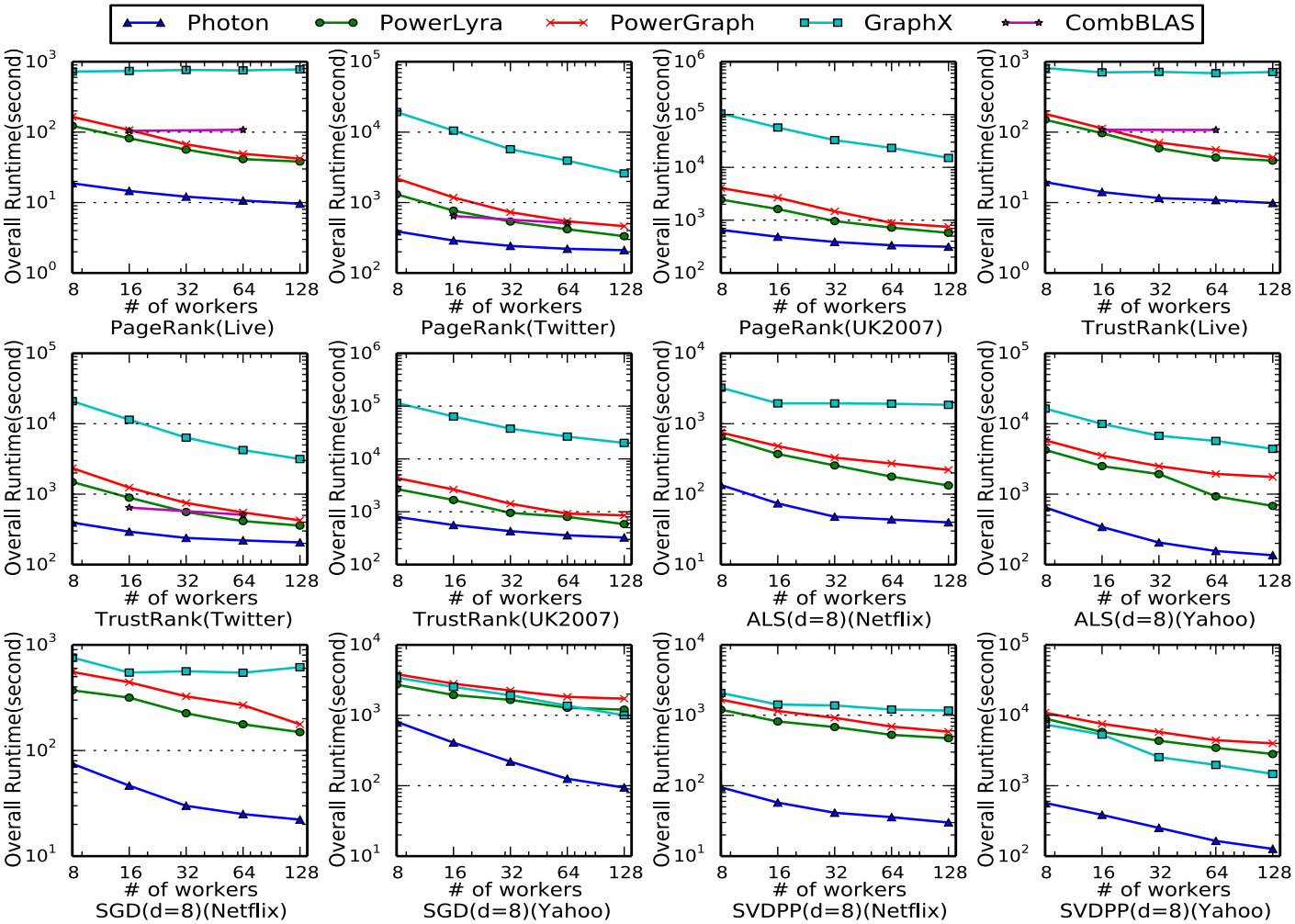




性能提升（Edge-Centric Execution Engine）



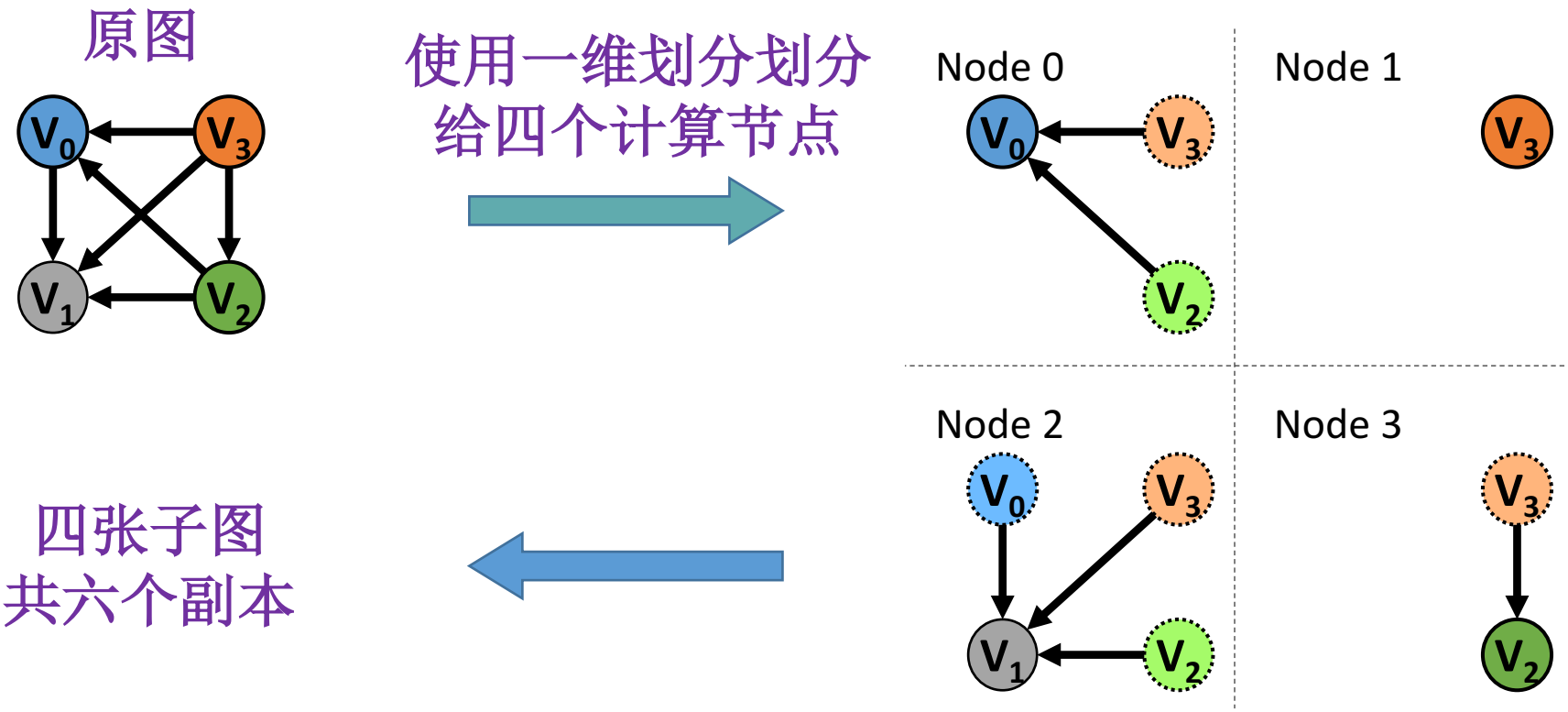
性能提升


Photon
PowerLyra
PowerGraph
GraphX
CombBLAS
Overall Runtime(second)
# of workers
PageRank(Live)
PageRank(Twitter)
PageRank(UK2007)
TrustRank(Live)
TrustRank(Twitter)
TrustRank(UK2007)
ALS(d=8)(Netflix)
ALS(d=8)(Yahoo)
SGD(d=8)(Netflix)
SGD(d=8)(Yahoo)
SVDPP(d=8)(Netflix)
SVDPP(d=8)(Yahoo)

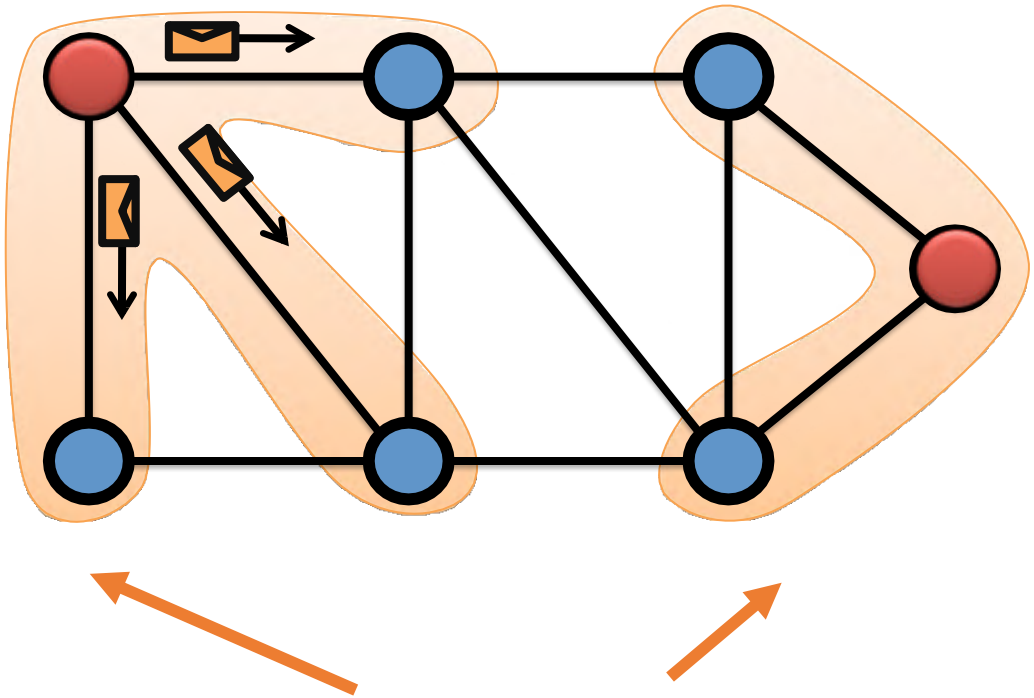
图的三维划分加速计算

研究现状：一维划分

一维划分：数据图被以点为粒度地划分给各个计算节点



研究现状：一维划分



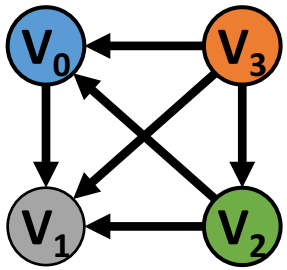
在一维划分的情况下，
任务划分的粒度也为点，
与数据划分的粒度相同

- **点程序**
  - 每一个点程序都可以读取或写入其对应点领域范围内的数据
- **通讯**
  - 通过消息传递 (e.g. Pregel)
  - 通过共享状态 (e.g. GraphLab)
- **并行**
  - 通过同时执行多个领域不相交的点程序
- **缺点**
  - 负载不均衡!!!

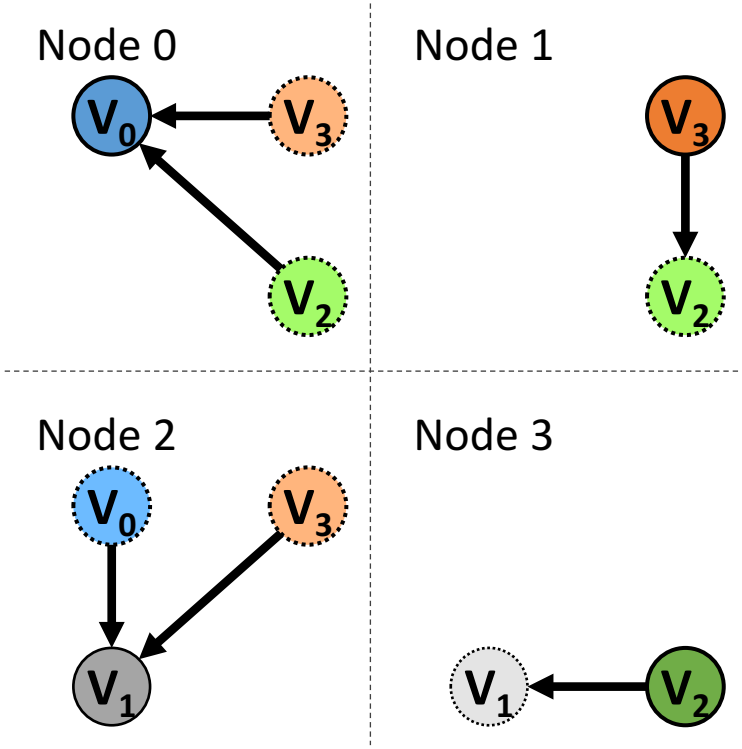
研究现状：二维划分

二维划分：数据图被以边为粒度地划分给各个计算节点

原图



使用二维划分划分给四个计算节点



四张子图
共六个副本

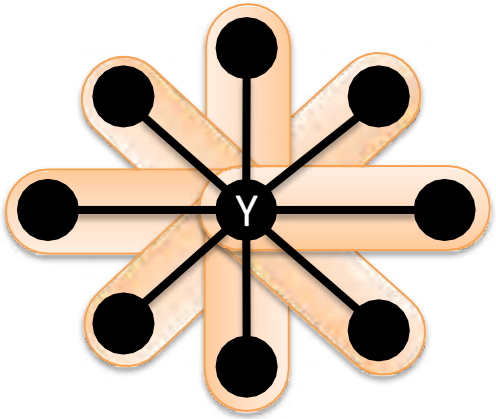
研究现状：二维划分

PowerGraph 的编程模型

Gather

用户自定义:

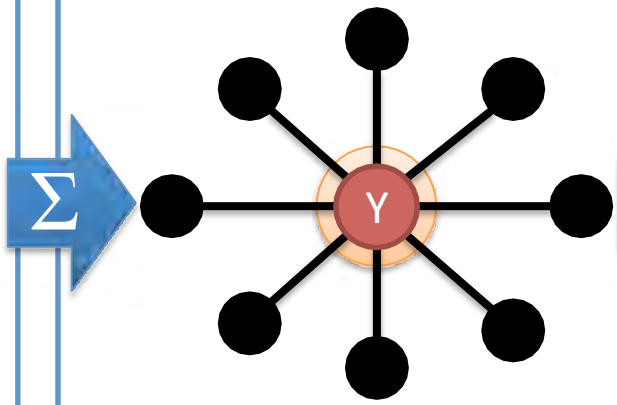
- Gather(Y) → Σ
- $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$



Apply

用户自定义:

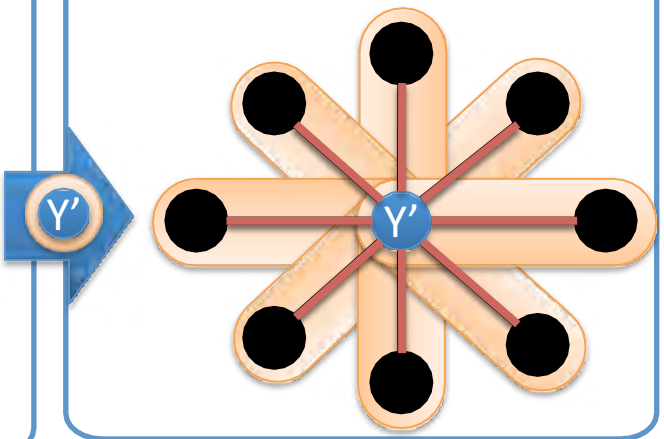
- Apply(Y , Σ) → Y'



Scatter

用户自定义:

- Scaber(Y') → —



在二维划分的情况下，任务划分的粒度也为边，
与数据划分的粒度相同

背景：图计算种类丰富

- 传统图分析类应用

  许多图应用目的在于
  分析图上的拓扑关系
  等图论性质

- 示例
  - 最短路径
  - 三角形个数
  - **PageRank**
  - 联通分量

背景：图计算种类丰富

传统图分析类应用

许多图应用目的在于分析图上的拓扑关系等图论性质

建模成图计算的机器学习应用

同时，许多机器学习和数据挖掘类的应用也可以被建模成图计算应用进行计算

示例

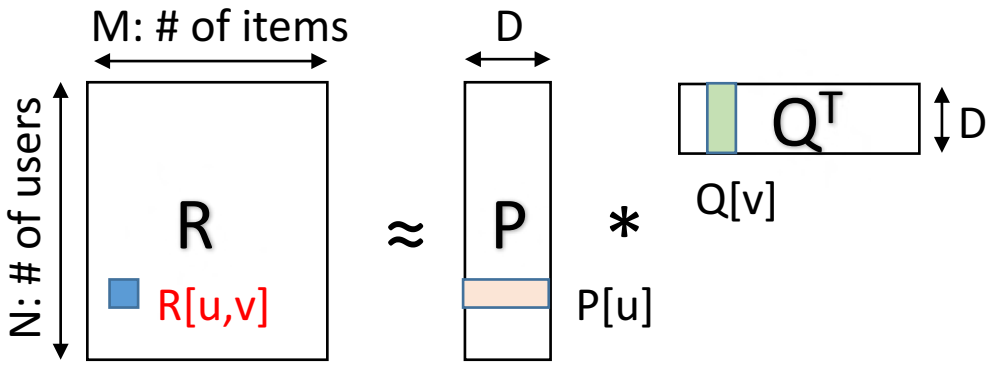
- 最短路径
- 三角形个数
- **PageRank**
- 联通分量

示例

- 协同过滤
- 稀疏矩阵相乘
- 神经网络
- 矩阵分解

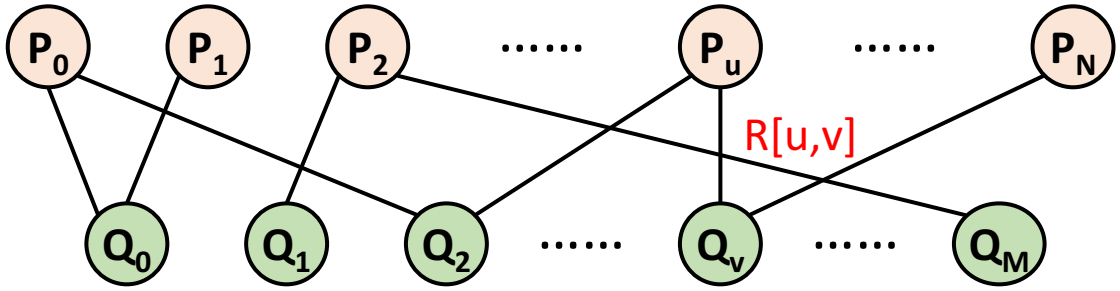
示例：协同过滤

通过矩阵描述



R[u,v] ≈ <P[u], Q[v]>

建模成图计算



协同过滤

通过给出的用户对物品的打分情况估测未给出的打分

建模成图计算后

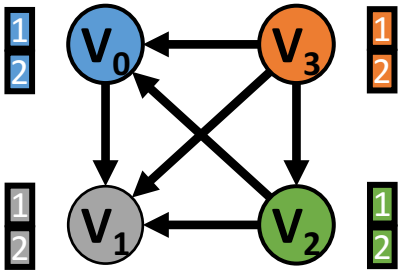
每一个点的点权为长度 **D** 的向量，是可再划分单元

这一现象是很多图计算应用共有的模式

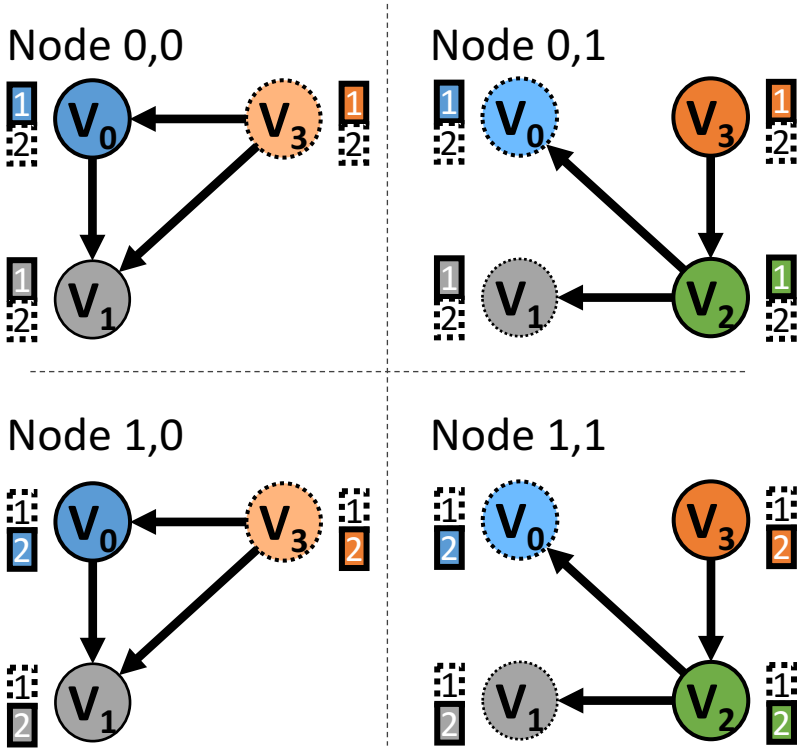
研究内容：三维划分

三维划分：将数据图中的点切分成子点，然后对由子点构成的图层进行二维划分

原图

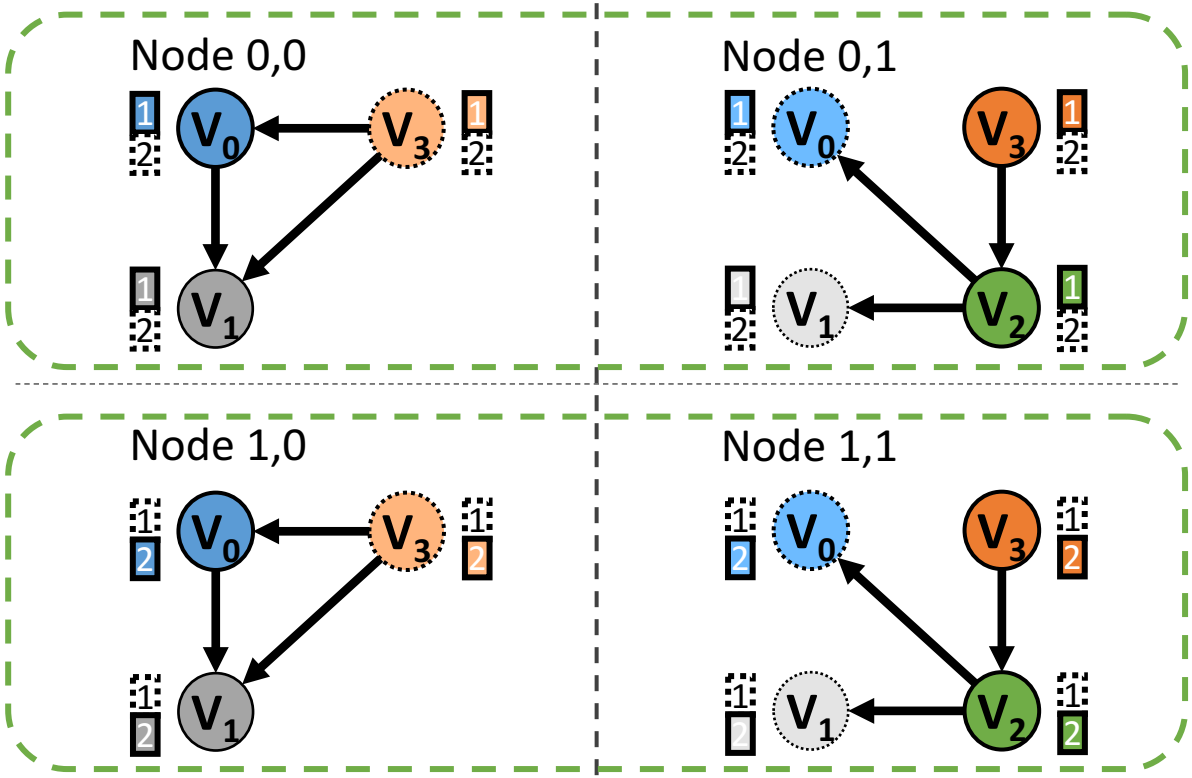


使用三维划分划分给四个计算节点



每层两张子图，三个副本

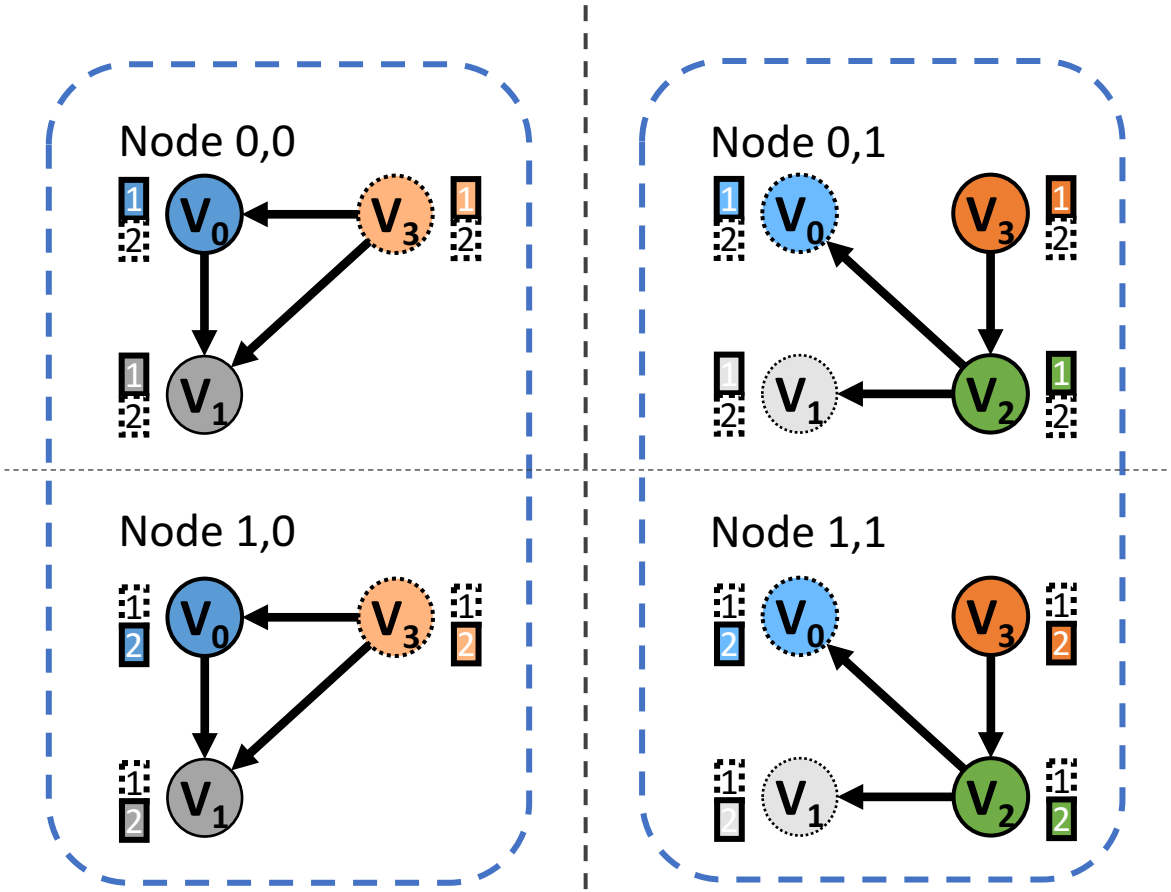
研究内容：两类通讯



层内通讯

- 更少的子图
- 更少的副本
- 更少的通讯！

研究内容：两类通讯



层内通讯

- 更少的子图
- 更少的副本
- 更少的通讯！

层间通讯

- 之前并不存在
- 不断增大！

小结

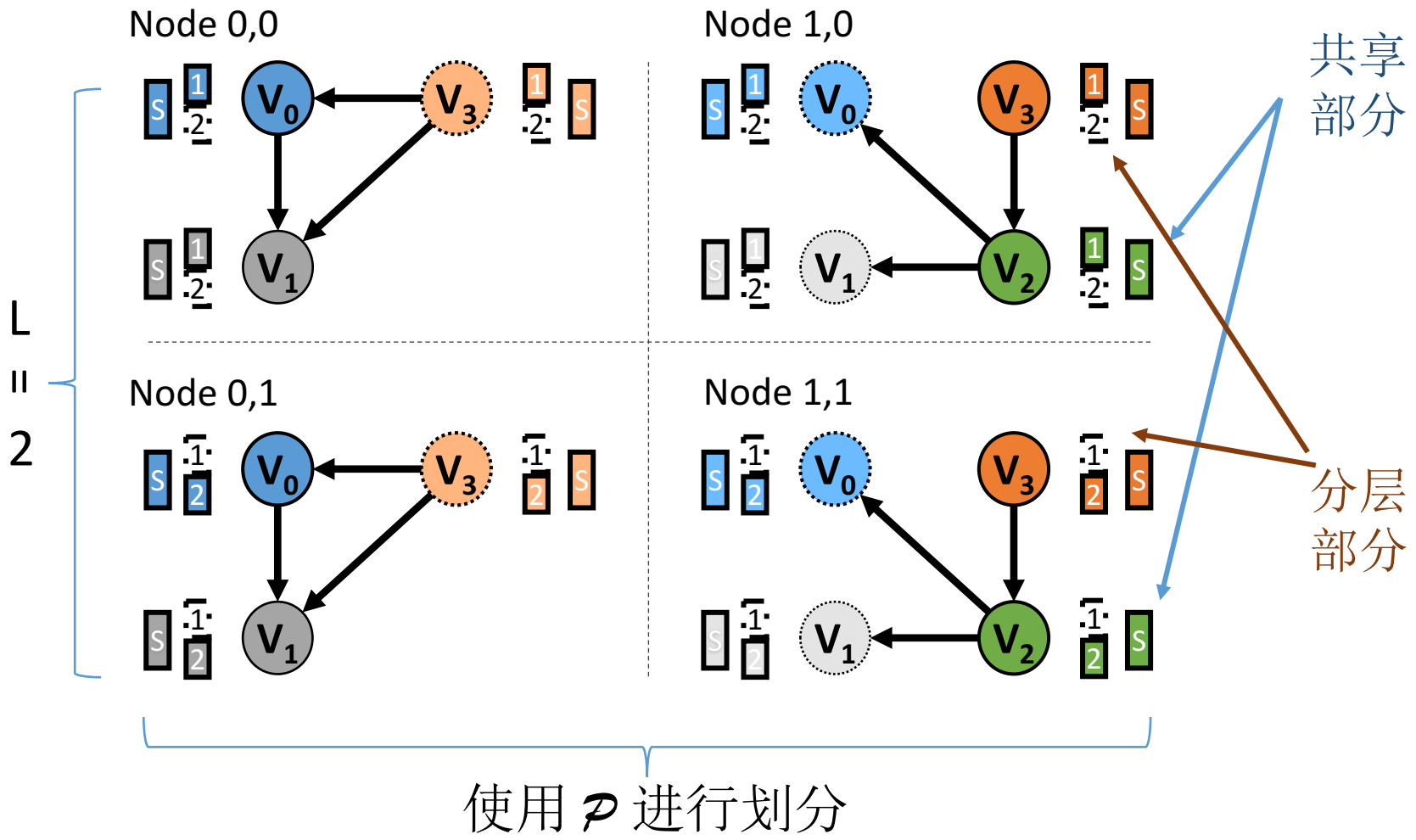
- 我们找到了一个新的划分图计算任务的维度
- 发现
  - 点权中的向量上进行的计算经常是对位地
  - 很容易并行化
    - 通过将下标相同的对位数据划分到同一个计算节点上，可以不引发任何通讯！

研究内容：新系统

- 现有的分布式图计算系统无法建模层间通讯
- CUBE
  - 采用三维划分
  - 基于新型的计算模型 UPPS
  - 使用矩阵执行引擎

研究内容：三维划分方法

三维划分方法 (L, $\mathcal{P}$): L 是层数，$\mathcal{P}$ 则是一个已有的二维划分方法



研究内容：矩阵执行引擎

CUBE

以点为中心

MAP

矩阵执行引擎

- ✓ 简单易用
- ✓ 与现有系统兼容
- ✖ 效率不高

- ✖ 编程复杂度高
- ✓ 高效
- ❖ 简单的寻址方法
- ❖ Hilbert order

我们通过自动地将以点为中心的程序转换成矩阵操作执行，同时拥有了两方的好处；该方法由 **GraphMat** 系统刷先提出。

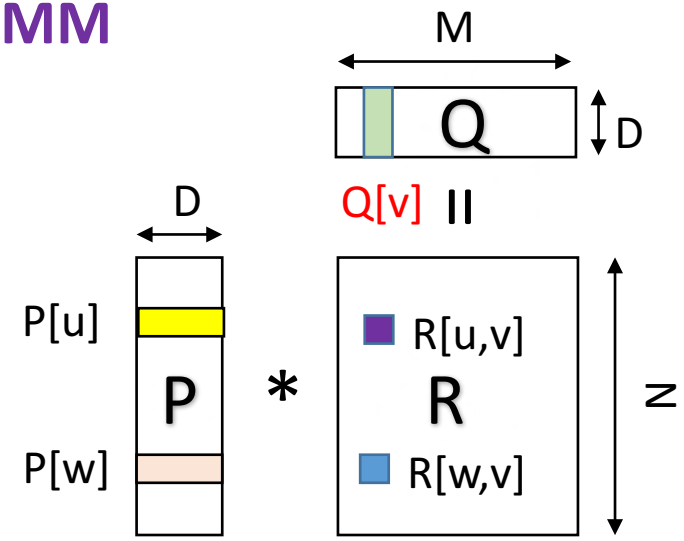
实验环境

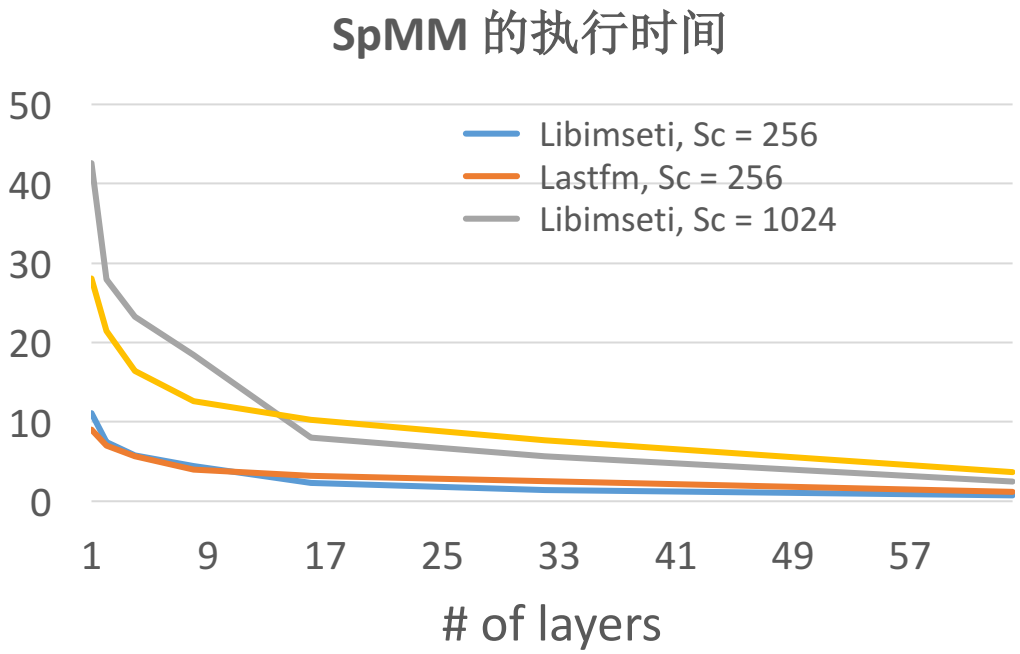
- **比较对象:** PowerGraph 和 PowerLyra
  - 缺省的二维划分方法 *oblivious* 被用于测试 PowerGraph
  - 相对的，PowerLyra 的测试枚举了所有种类的划分方法
  - CUBE 系统使用的 $\mathcal{P}$ 与 PowerLyra 所用的方法保持一致
- **实验平台:** 八个计算节点; 通过 1Gb 以太网互联
- **数据集**

代号	|U|	|V|	|E|	最好的二维划分方法
Libimseti	135,359	168,791	17,359,346	Hybrid-cut
Last.fm	359,349	211,067	17,559,530	Bi-cut
Netflix	17,770	480,189	100,480,507	Bi-cut

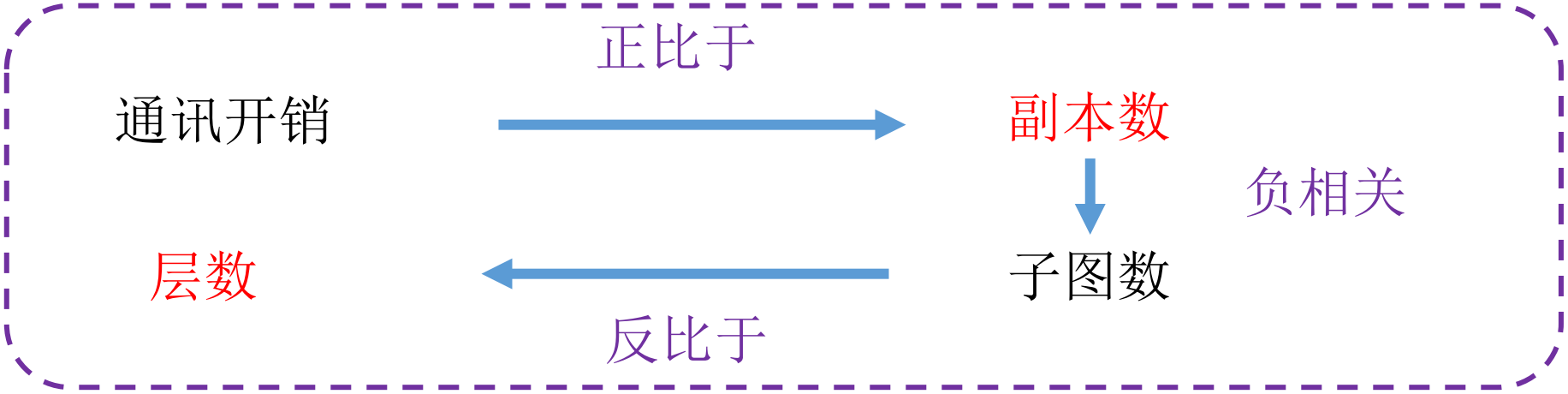
微型测试集：SpMM

SpMM

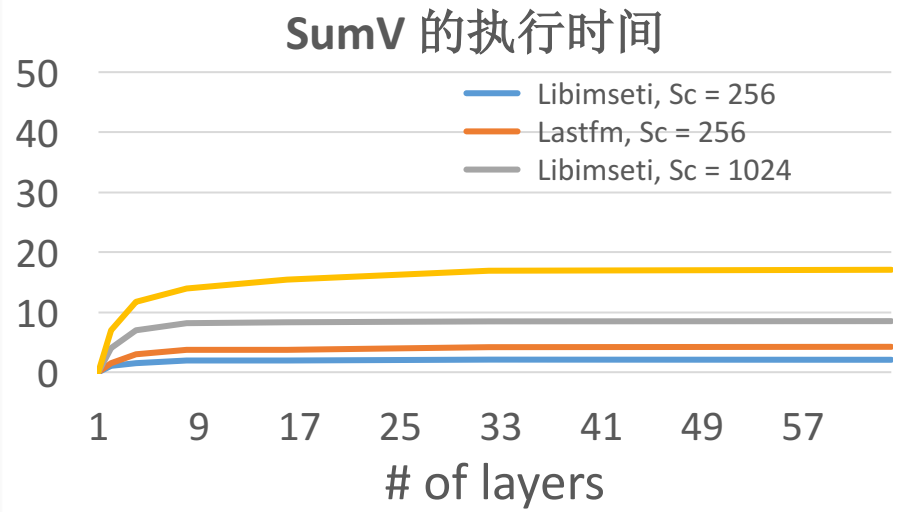


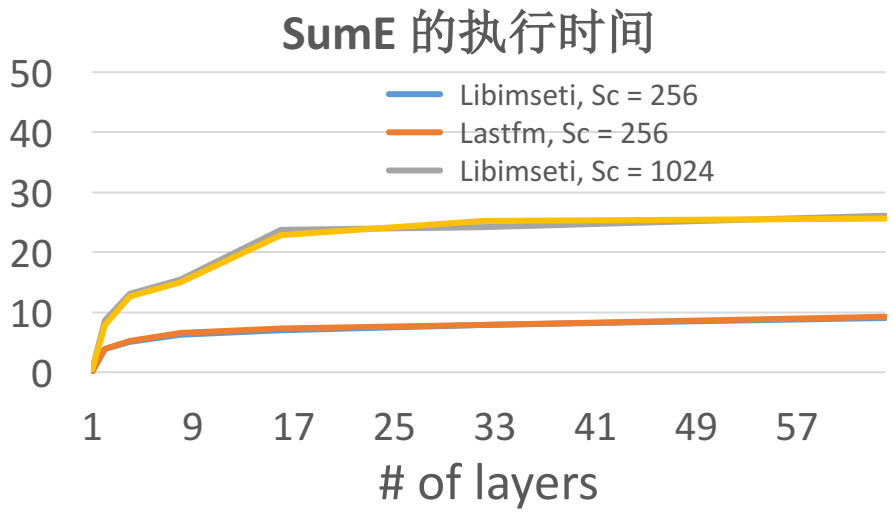


$Q[v] = R[u,v]*P[u] + R[w,v]*P[w]$



微型测试集：SumV & SumE





SumV: 对点权向量累和

SumE: 对边权向量累和

通讯开销 正比于 $\frac{L-1}{L}$, L 为层数

测试结果：PowerLyra & PowerGraph

CUBE

数据集	D	执行时间					
		GD			ALS		
		PowerGraph	PowerLyra	CUBE	PowerGraph	PowerLyra	CUBE
Libimseti	64	6.82	6.89	2.59 (4)	87.0	86.8	28 (64)
	128	11.64	11.62	3.33 (8)	331	331	109 (64)
Lastfm	64	10.4	9.86	2.48 (4)	158	111	57 (64)
	128	18.6	17.8	3.47 (8)	Failed	Failed	230 (64)
Netflix	64	18.3	7.42	4.16 (1)	179	66.0	42.5 (8)
	128	30.6	11.3	6.55 (2)	Failed	239	118 (8)

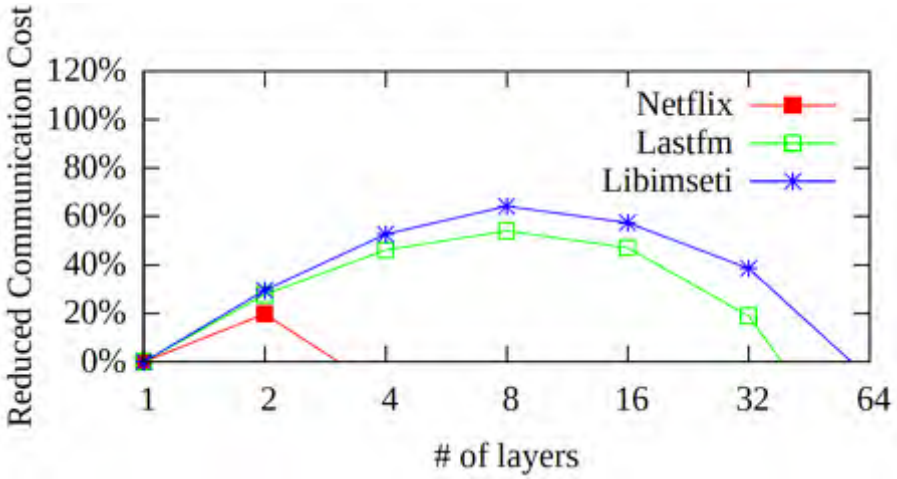
✓ 较之 PowerLyra 提速最高 4.7 倍

✓ 较之 PowerGraph 提速最高 7.3 倍

测试结果：通讯量减少

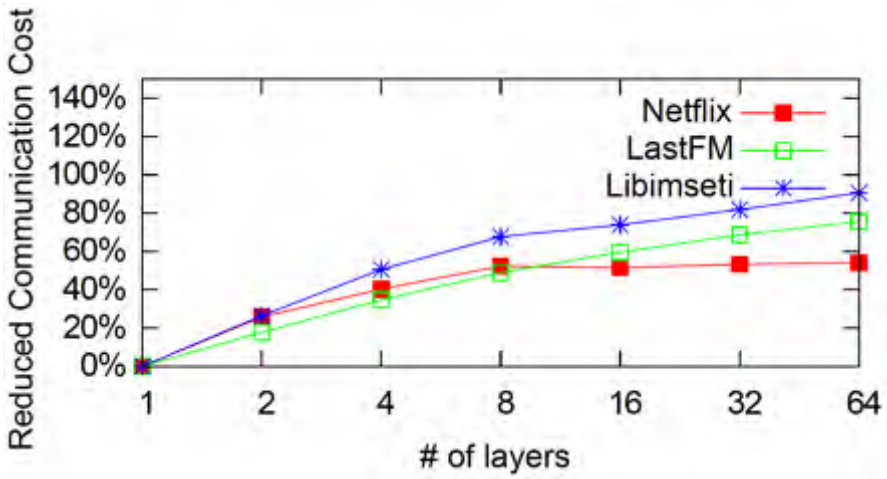
CUBE

GD 算法上的通讯量减少



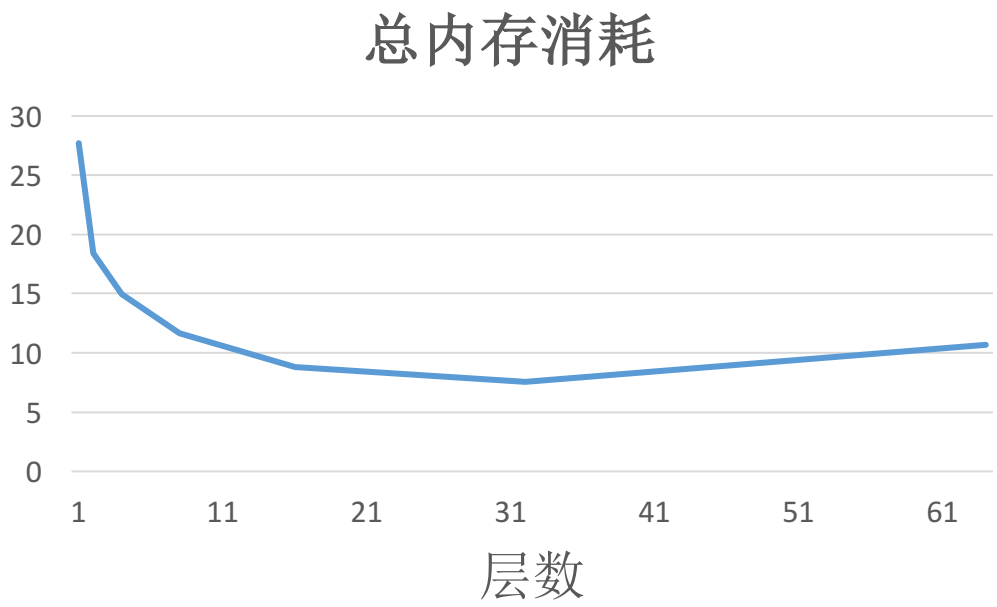
- ✓ 对于 Lastfm 和 Libimseti 数据集一半的加速来自于通讯减少
- ✖ 对 Netflix 数据集效果不佳
- ❖ 如果设定 D 为 2048，即使是对 Netflix 数据集也能提速 2.5 倍

ALS 算法上的通讯量减少



- ✓ 几乎全部的加速来自于通讯减少.
- ❖ ALS 的计算需要执行复杂度为 $O(N^3)$ 的计算单元

测试结果：内存消耗



在 64 个计算节点上执行 ALS 算法的总内存消耗，其中 D=32, $Sc=D^2+D=1056$

内存消耗

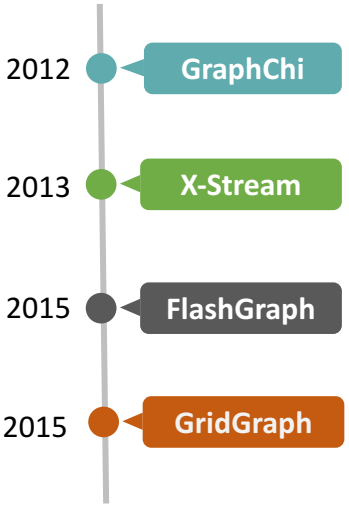
- 比 L=1 时要小
- 最佳的执行时间不一定同时有最小的内存开销

外存图计算的加速

单机的外存图计算

现有的单机的计算规模已经可以处理大规模的图（这种情况下，磁盘的IO的操作将成为瓶颈）

单机的图计算系统



Wrong trade-off: They improve the disk I/O locality at the cost of **increasing the total amount of disk I/O.**

Our work

Clip: 新型单机图计算系统

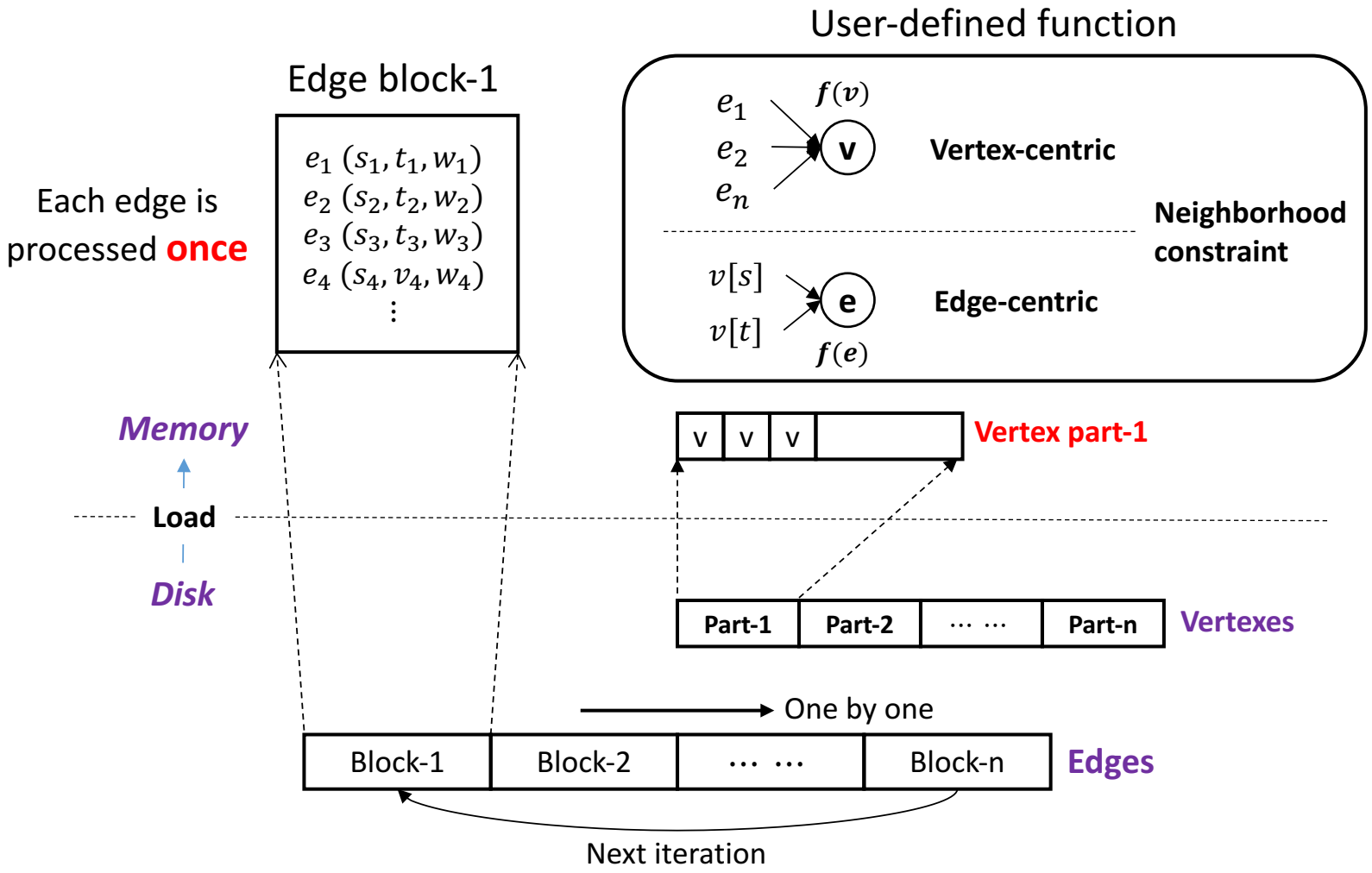
特征

- More flexible processing order
- Breaking the neighborhood constraint
- More efficient algorithms can be used

Primary concern

Reducing the total amount of disk I/O

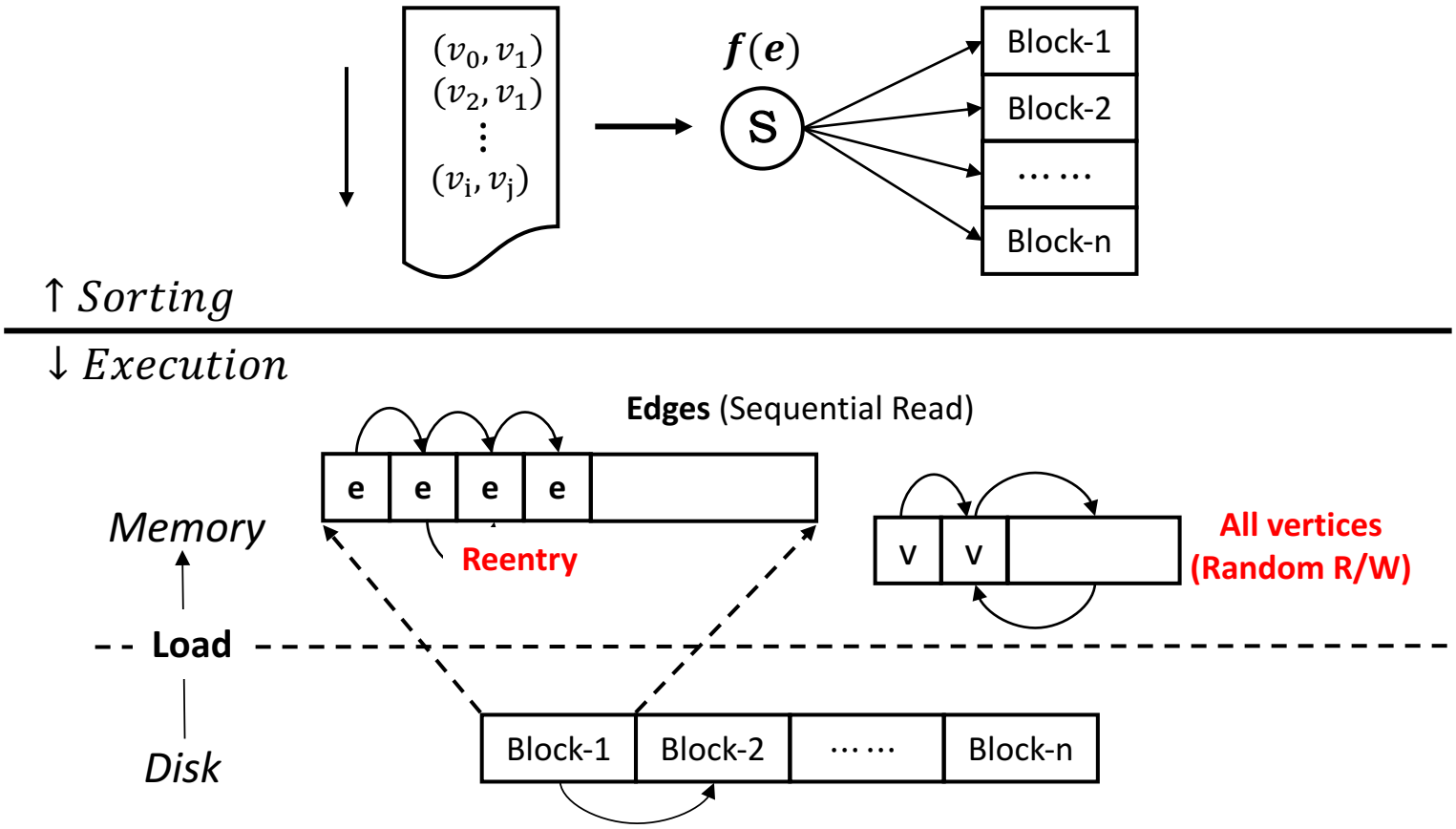
现有工作的限制



每一条边都只被处理一次，之后就被换出
用户定义的函数只能作用在邻边上和邻接点上

增加了循环的次数
增加了IO的数目，降低性能

解决的方法



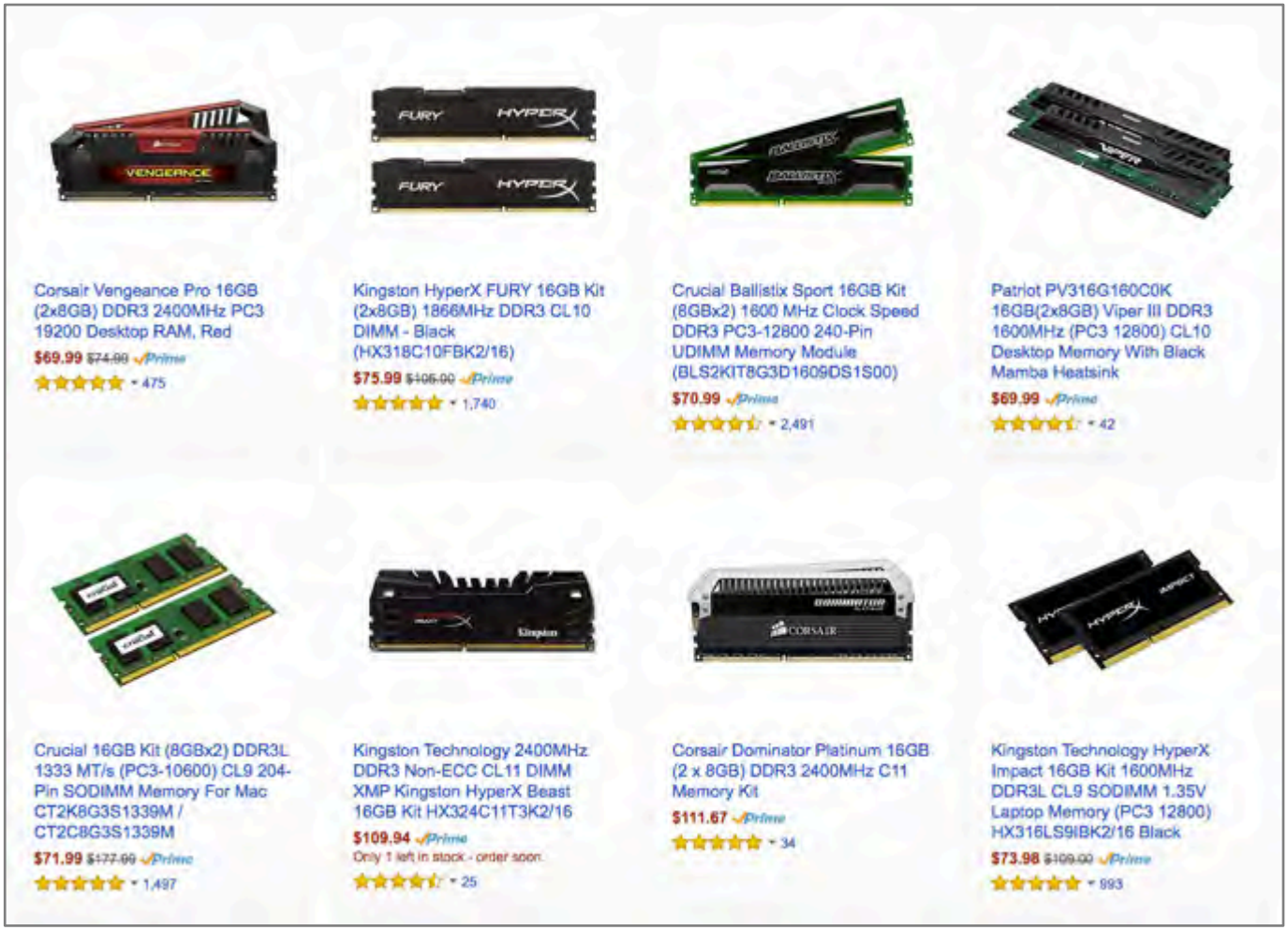
CLIP: Squeezing out all the value of loaded data

装载所有节点数据到内存，尽可能多使用已经装载到内存中的所有数据

1 尽可能在装载的数据上多次进行计算

2 更新节点数据的时候，可以更新任何在内存中的节点的数据

节点信息存放在内存中？



From https://www.amazon.com/

典型的图数据集的大小情况

The real-world graph datasets

	Vertexes	Edges	Vertexes Size	Edges Size
LiveJournal	4.85M	69.0M	37MB	0.53GB
Dimacs	23.9M	58.3M	183MB	0.67GB
Twitter	41.7M	1.47B	317MB	10.9GB
Friendster	65.6M	1.8B	501MB	13.5GB
Yahoo	1.4B	6.64B	10.5GB	49.4GB
Twitter-Big	288M	60B	2.25GB	480GB
Facebook-Big	1.39B	400B	10.4G	3.1TB

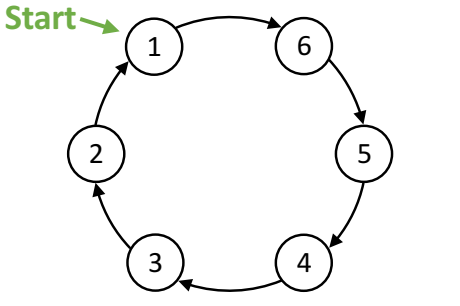
The biggest real-world graph:

- The edges size are much larger than vertexes size
- 32G memory is sufficient for vertexes size
- 32G memory cannot hold all edges
- 32G DDR3 chips are very cheap and prevalent now

Tips: Some single-machine graph systems have similar design, such as GraphChi and FlashGraph

计算方式的改进

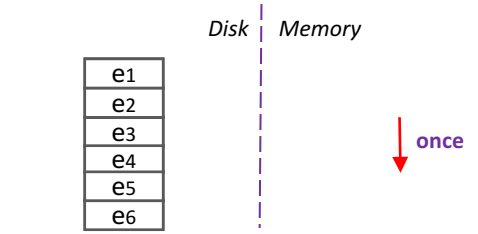
Example: Calculating single source shortest path (SSSP)



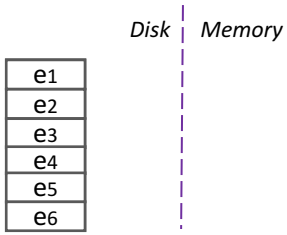
	source	target	weight
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

Edge list on disk

Prior system: process once

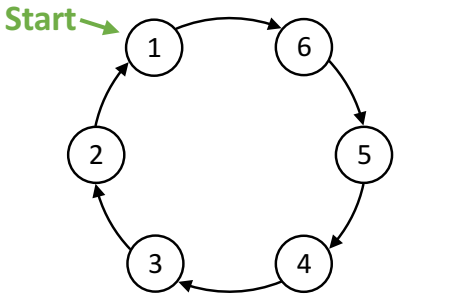


Clip: process multiple times



计算方式的改进

Example: Calculating single source shortest path (SSSP)

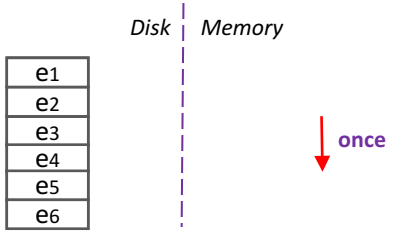


	1					6
Init dist	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

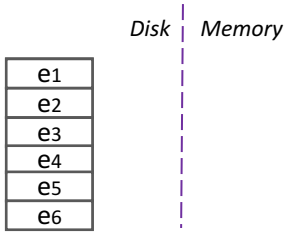
Edge list on disk

Prior system: process once



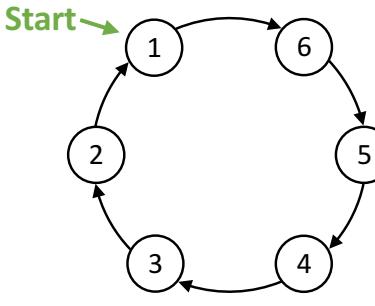
	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1
Iteration 2	0	∞	∞	3	2	1

Clip: process multiple times



计算方式的改进

Example: Calculating single source shortest path (SSSP)

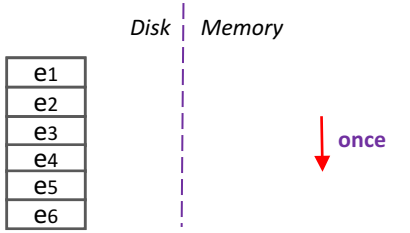


	1				6
Init dist	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

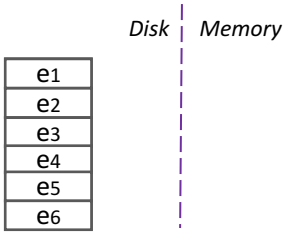
Edge list on disk

Prior system: process once



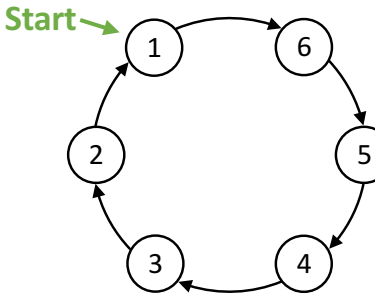
	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1
Iteration 2	0	∞	∞	3	2	1
Iteration 3	0	∞	4	3	2	1

Clip: process multiple times



计算方式的改进

Example: Calculating single source shortest path (SSSP)

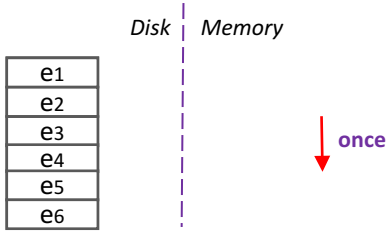


	1	6	
Init dist	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

Edge list on disk

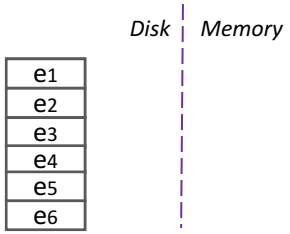
Prior system: process once



	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1
Iteration 2	0	∞	∞	3	2	1
Iteration 3	0	∞	4	3	2	1
Iteration 4	0	5	4	3	2	1

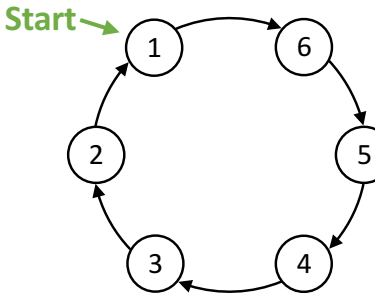
$Total\ amount: 4 * 6 * sizeof(e)$

Clip: process multiple times



计算方式的改进

Example: Calculating single source shortest path (SSSP)

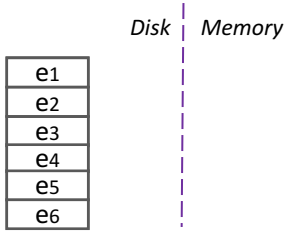


	1	∞	∞	∞	∞	6
Init dist	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

Edge list on disk

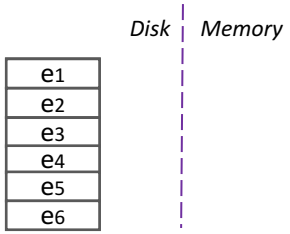
Prior system: process once



	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1
Iteration 2	0	∞	∞	3	2	1
Iteration 3	0	∞	4	3	2	1
Iteration 4	0	5	4	3	2	1

$Total\ amount: 4 * 6 * sizeof(e)$

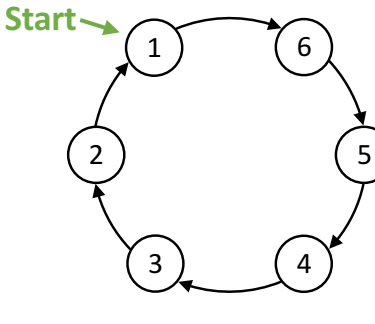
Clip: process multiple times



	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	3	2	1

计算方式的改进

Example: Calculating single source shortest path (SSSP)

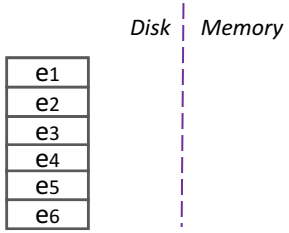


	1	⟶	6
Init dist	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

Edge list on disk

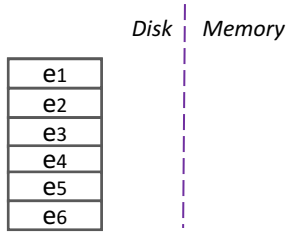
Prior system: process once



	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1
Iteration 2	0	∞	∞	3	2	1
Iteration 3	0	∞	4	3	2	1
Iteration 4	0	5	4	3	2	1

$Total\ amount: 4 * 6 * sizeof(e)$

Clip: process multiple times



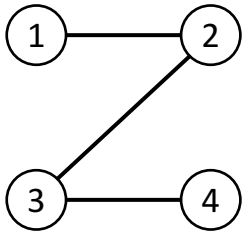
Pass two

	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	3	2	1
Iteration 2	0	5	4	3	2	1

$Total\ amount: 2 * 6 * sizeof(e)$

计算方式的改进

Example: Calculating weakly connected component (WCC)

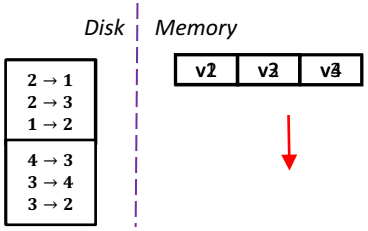


Undirected graph

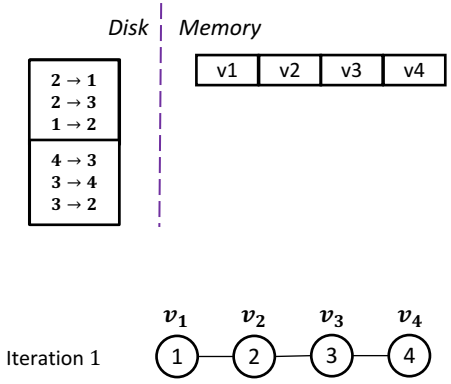
	source	*target*
e_1	2	3
e_2	2	1
e_3	1	2
e_4	4	3
e_5	3	4
e_6	3	2

Edge list on disk

Prior system: label propagation based

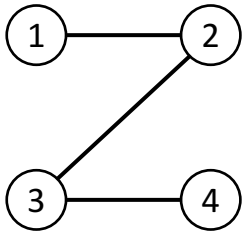


Clip: disjoint set (all vertexes in memory)



计算方式的改进

Example: Calculating weakly connected component (WCC)

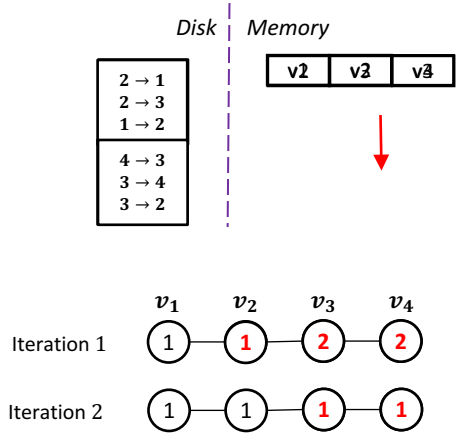


Undirected graph

	source	*target*
e_1	2	3
e_2	2	1
e_3	1	2
e_4	4	3
e_5	3	4
e_6	3	2

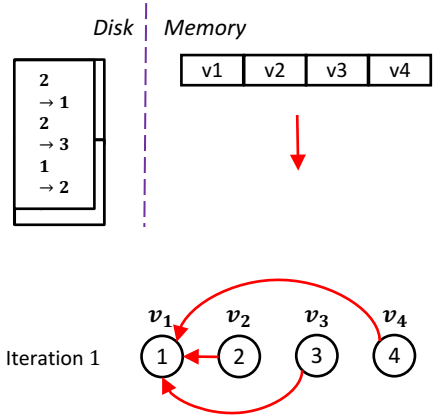
Edge list on disk

Prior system: label propagation based



***Total amount*: $2 * 6 * sizeof(e)$**

Clip: disjoint set (all vertexes in memory)



***Total amount*: $1 * 6 * sizeof(e)$**

编程模型

Programming model of CLIP

$Sort(\mathcal{F}_s)$ — $\mathcal{F}_s$:= **double** function(*Edge* &*e*)

$Exec(\mathcal{F}_e)$ — $\mathcal{F}_e$:= **void** function(*Vertexes* &*v_list*, *Edge* &*e*)

$VMap(\mathcal{F}_v)$ — $\mathcal{F}_v$:= **void** function(*Vertexes* &*v_list*, *VertexID* &*vid*)

All vertexes

Algorithm 2 SSSP Algorithm in CLIP.

```
Functions:
  F_v(v_list, vid) :— {
    if vid == start do
      v_list[vid].dist ← 0; v_list.setActive(vid,true);
    else v_list[vid].dist ← INF; v_list.setActive(vid, false); }
  F_e(v_list, e) :— {
    if v_list[e.dst].dist > v_list[e.src].dist + e.weight do
      v_list[e.dst].dist ← v_list[e.src].dist + e.weight;
      v_list.setActive(e.dst,true);
    else v_list.setActive(e.dst, false); }
Computation:
  VMap(F_v);
  Until convergence:
    Exec(F_e);
```

Algorithm 3 WCC Algorithm in CLIP.

```
Functions:
  F_find(v_list, vid) :— {
    if v_list[vid].pa == vid do return vid;
    else return v_list[vid].pa = F_find(v_list, v_list[vid].pa); }
  F_union(v_list, src, dst) :— {
    s ← F_find(v_list, src);
    d ← F_find(v_list, dst);
    if s < d do v_list[d].pa ← v_list[s].pa;
    else if s > d do v_list[s].pa ← v_list[d].pa; }
  F_e(v_list, e) :— { F_union(v_list, e.src, e.dst); }
  F_v(v_list, vid) :— {
    v_list[vid].pa ← vid; v_list.setActive(vid,true); }
Computation:
  VMap(F_v);
  Exec(F_e);
```

编程模型的效果

The user only write a very short code———SSSP on CLIP vs SSSP by user

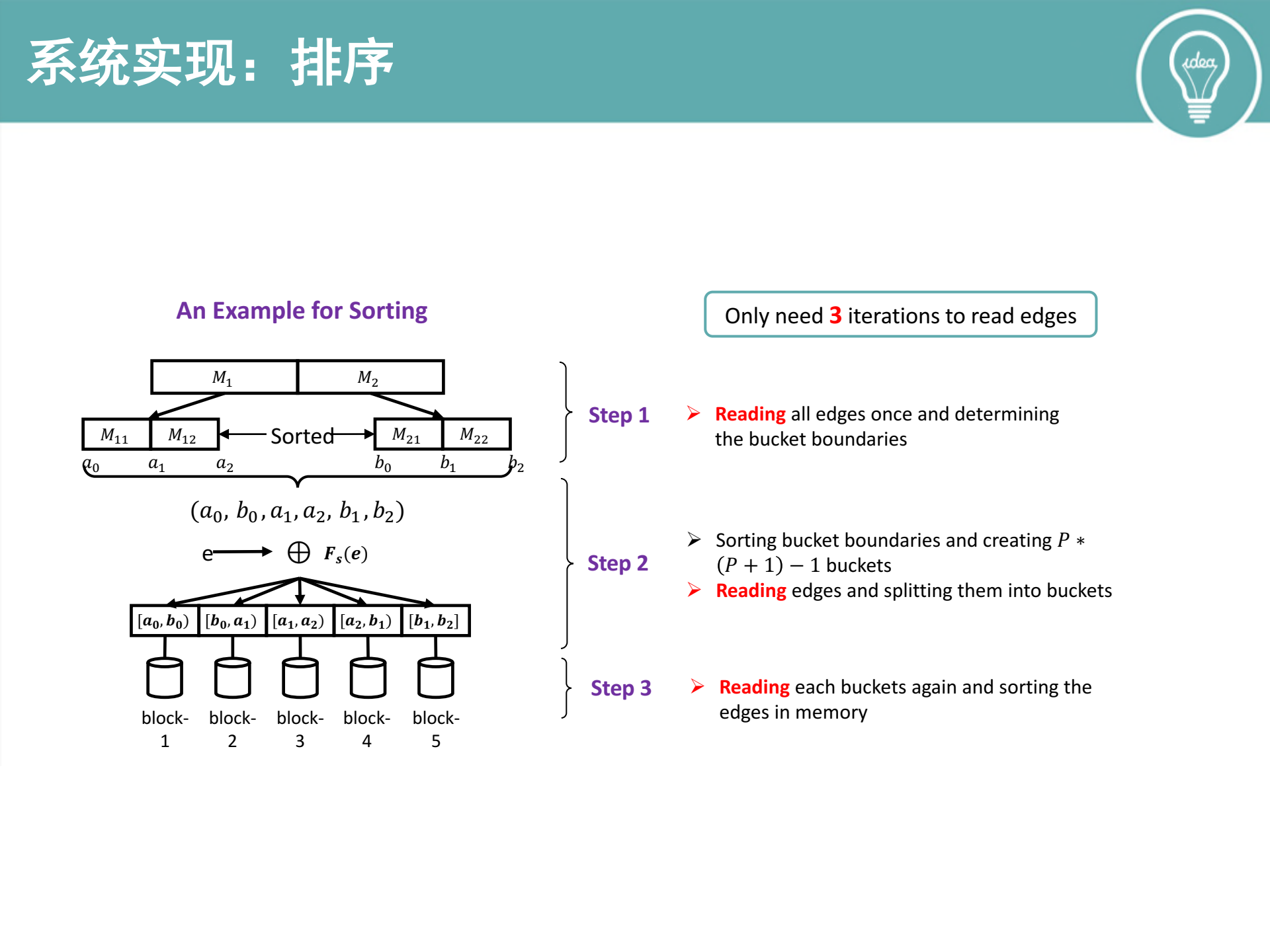
CLIP: **50 lines of code** are enough

Sorting: 440 lines

Exec: 614 lines

The user need to write more than **1000 lines of code** to implement the optimized SSSP algorithm, which consists of **sorting, overlapping, multi-thread…**

系统实现：排序



An Example for Sorting

Only need **3** iterations to read edges

Step 1

- **Reading** all edges once and determining the bucket boundaries

Step 2

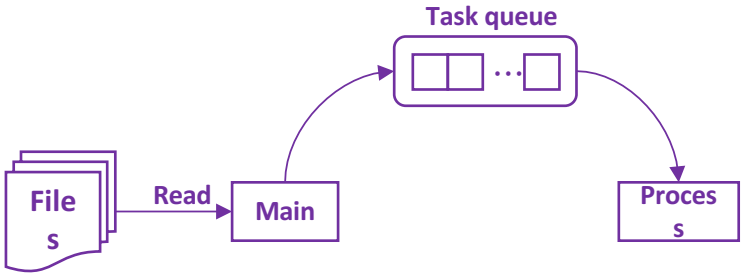
- Sorting bucket boundaries and creating $P * (P + 1) - 1$ buckets
- **Reading** edges and splitting them into buckets

Step 3

- **Reading** each buckets again and sorting the edges in memory

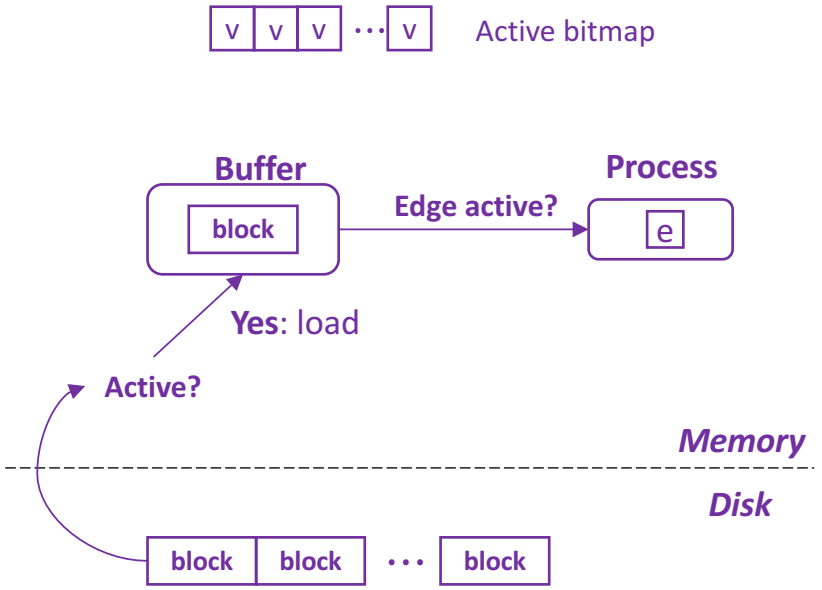
系统实现：优化

- Disk I/O—Overlapping



- Concurrency control (Multi-thread)
  - Neighborhood constraint: fined-grained locking
  - Many iterative algorithms can tolerate update overwriting: SSSP (Hogwild[NIPS'11])

- Selective scheduling



实验设置

The real-world graph datasets

	Vertexes	Edges	Vertexes Size	Edges Size
LiveJournal	4.85M	69.0M	37MB	0.53GB
Dimacs	23.9M	58.3M	183MB	0.67GB
Twitter	41.7M	1.47B	317MB	10.9GB
Friendster	65.6M	1.8B	501MB	13.5GB
Yahoo	1.4B	6.64B	10.5GB	49.4GB

Tips: Dimacs and Yahoo have a large diameter

Test environment

Single machine

- **CPU:** Two Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz (each has 8-cores)
- **DRAM:** 32GB, L3 cache 20MB
- **Disk:** Standard 1TB SSD drive. The average throughput is about 250MB/s for sequential read.

Test benchmarks

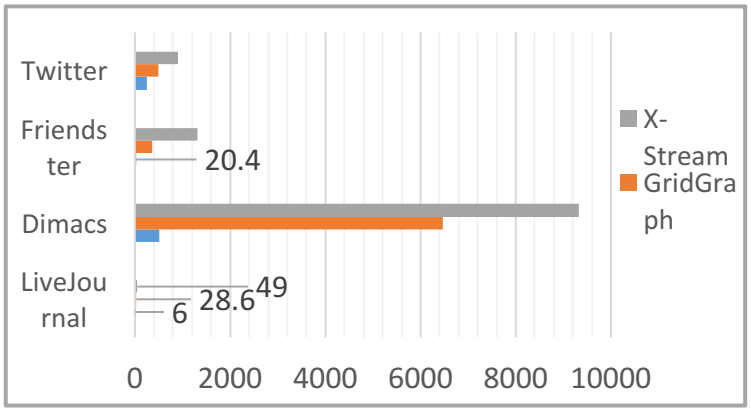
- **Relaxing-based applications**
  - SSSP (Single Source Shortest Path)
  - BFS (Breadth-first Search)
- **Beyond-neighborhood applications**
  - WCC (Weakly connected component)
  - MIS (Maximal Independent Set)
  - MCST (Minimum Cost Spanning Tree)
- **Beyond the Neighborhood**
  - PageRank
  - SpMV (Multiply the sparse adjacency matrix of a directed graph with a vector of values, one per vertex)

数据复用的效果

Execution time (in seconds)

		Out-Of-Core					In-Memory				
		LiveJournal	Dimacs	Friendster	Twitter	Yahoo	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
SSSP	X-Stream	118.4	22647	3346	2255	>86400	8.45	853.2	-	-	-
	GridGraph	48.1	13480	784.6	1083	17432	6.97	889.9	85.31	83.51	-
	CLIP	11.23	1981	55.79	600.6	6932	5.09	316.1	55.85	91.82	-
BFS	X-Stream	22.94	6538	1084	627.4	>86400	4.06	114.9	-	-	-
	GridGraph	15.4	5239	493.7	209.6	7403	2.49	406.2	61.54	32.16	-
	CLIP	5.46	1059	38.55	110.4	3297	2.53	96.12	38.72	44.7	-

I/O amount of SSSP (GB)



Iteration Count

	LiveJornal	Dimacs	Friendster	Twitter	Yahoo
SSSP					
X-Stream	45	10790	50	39	>500
GridGraph	37	10788	27	33	5824
CLIP	8	934	1	19	484
BFS					
X-Stream	16	6263	30	14	>720
GridGraph	15	6262	29	13	4375
CLIP	4	574	1	5	243

SSSP speedup: 1.8x-14.06x

BFS speedup: 1.9x-12.08x

更新更多节点

Execution time (in seconds)

		Out-Of-Core					In-Memory				
		LiveJournal	Dimacs	Friendster	Twitter	Yahoo	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC	X-Stream	57.77	6751	2341	1194	>86400	10.25	185.3	-	-	-
	GridGraph	13.8	5757	466.6	272.3	2916	3.57	422.5	82.95	62.30	-
	CLIP	2.40	1.35	65.48	49.03	220.9	2.43	1.33	64.56	48.85	-
MIS	X-Stream	152.6	41.42	4867	3042	>86400	13.06	5.95	-	-	-
	GridGraph	122.1	39.19	3777	2473	>86400	2.98	14.46	253.7	156.1	-
	CLIP	2.57	1.17	62.49	49.08	220.2	2.58	1.21	62.18	49.13	-
MCST	X-Stream	34.21	15.89	1322	956.4	>86400	17.86	10.8	-	-	-
	CLIP	3.69	1.59	93.05	74.92	686.2	3.64	1.61	-	-	-

Iteration count

	LiveJornal	Dimacs	Friendster	Twitter	Yahoo
WCC					
X-Stream	13	6263	24	16	>360
GridGraph	7	6261	12	10	1368
CLIP	1	1	1	1	1
MIS					
X-Stream	54	31	65	53	>310
GridGraph	53	30	64	52	>400
CLIP	1	1	1	1	1
MCST					
X-Stream	6	10	6	4	>40
CLIP	1	1	1	1	1

CLIP

- Sequential implementation
- But only need **one** iteration

WCC speedup: 5.6x-4264x

MIS speedup: 34x-60x

MCST speedup: 9x-14x

更新更多节点

关于排序的测试
排序被很多应用所需要

排序包括了对于节点的排序
以及包括对于边的排序

Preprocessing time (seconds)

	GridGraph	CLIP (by source ID)	CLIP (by weight)
LiveJornal	4.57	5.06	9.06
Dimacs	4.09	5.12	6.81
Friendster	185.5	145.3	228.9
Twitter	160.3	126.2	188.1
Yahoo	1616	1410	1563

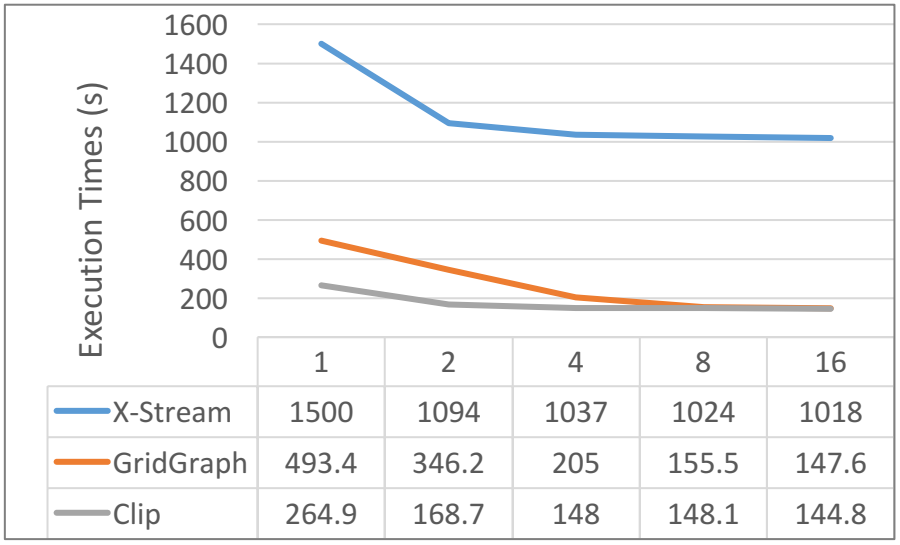
Total time on Friendster (seconds)

		Proc.	Exec.	Total
MIS	X-Stream	0	4867	4867
	GridGraph	185.5	3777	3962.5
	Clip	**145.3**	**62.49**	**207.79**
MCST	X-Stream	0	1322	1322
	Clip	**228.9**	**93.02**	**321.92**

MIS speedup: 19.07x　　**MCST speedup:** 4.1x

5.4 Evaluation—Numeric Applications

Execution time on changed iterations



	1	2	4	8	16
X-Stream	1500	1094	1037	1024	1018
GridGraph	493.4	346.2	205	155.5	147.6
Clip	264.9	168.7	148	148.1	144.8

Tips: Running 5 iterations of PageRank on Twitter Graph

PageRank and SpMV cannot be optimized by Clip

They have a fixed number of iterations

Performance comparison

Better performance

- Clip gives a better performance than GridGraph when the number of threads less than **8**.

Overlapping the disk readings and computation

Not bad

- Clip has a same performance with GridGraph when the number of threads more than 8.

使用体系结构相关方法优化图计算

- 图计算由于其应用的广泛以及规模的扩展，现在仍然是热点的研究内容
- 通过体系结构相关的方法可以加速图计算的运行
  - 使用体系结构局部性加速图计算（IEEE Transactions on Computers）
  - 图的三维划分加速计算(USENIX OSDI 2016)
  - 外存图计算的加速方法(USENIX ATC 2017)
- 图计算的不同模式，需要不同的加速方法

谢谢

更多内容请访问

madsys.cs.tsinghua.edu.cn