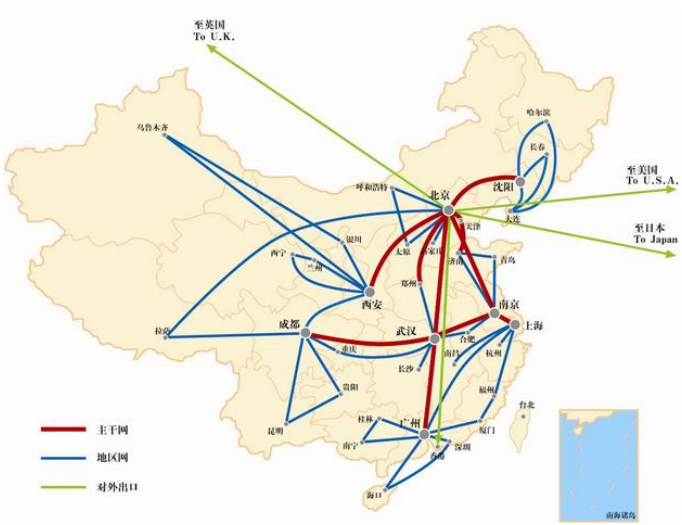
Introduction to Big Data Systems

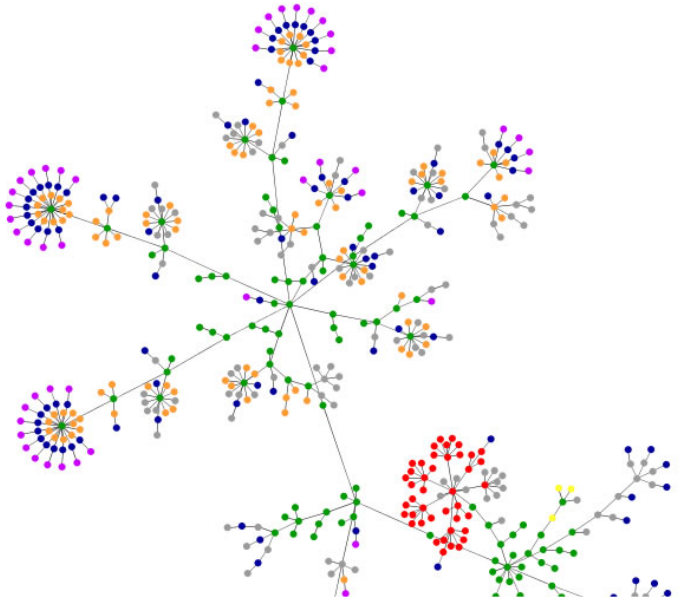
Lecture 6.Distributed Graph Computing

Wenguang CHEN
Tsinghua University

Graph Computation

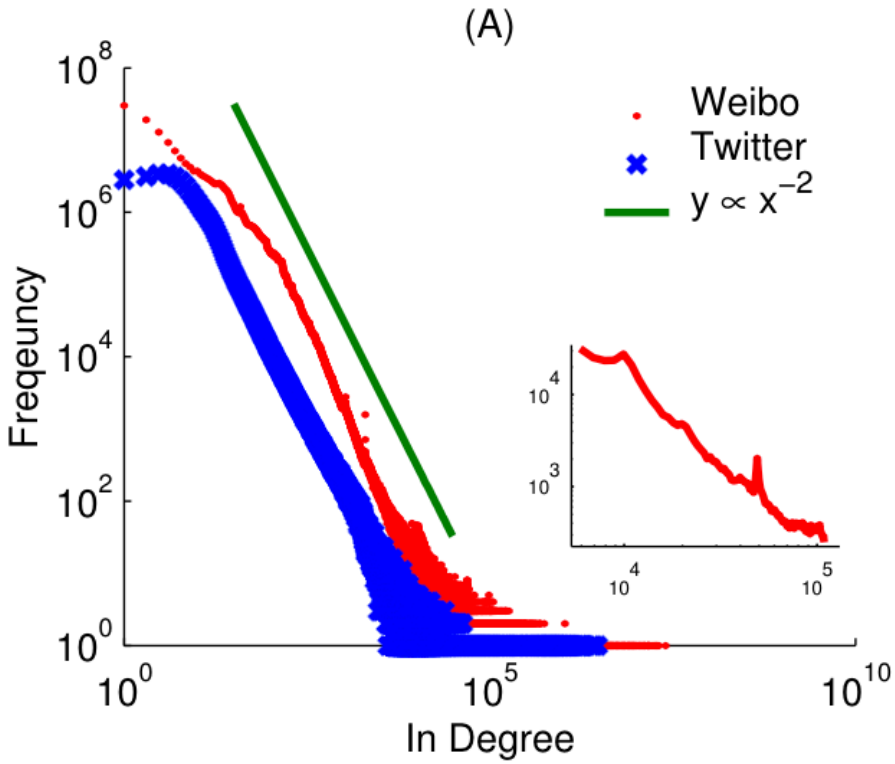
- Graph is one of the most general data structure and has many important applications
  - Twitter / Facebook / Weibo
  - Amazon user-item rating matrix
  - Bioinformatic / astrophysics / ..





Challenges in graph computing

- Large scale
  - Billions vertices, trillions of edges
- Poor locality
  - Random access on vertices
- Irregular topology
  - Power-law distributio in real-world graphs



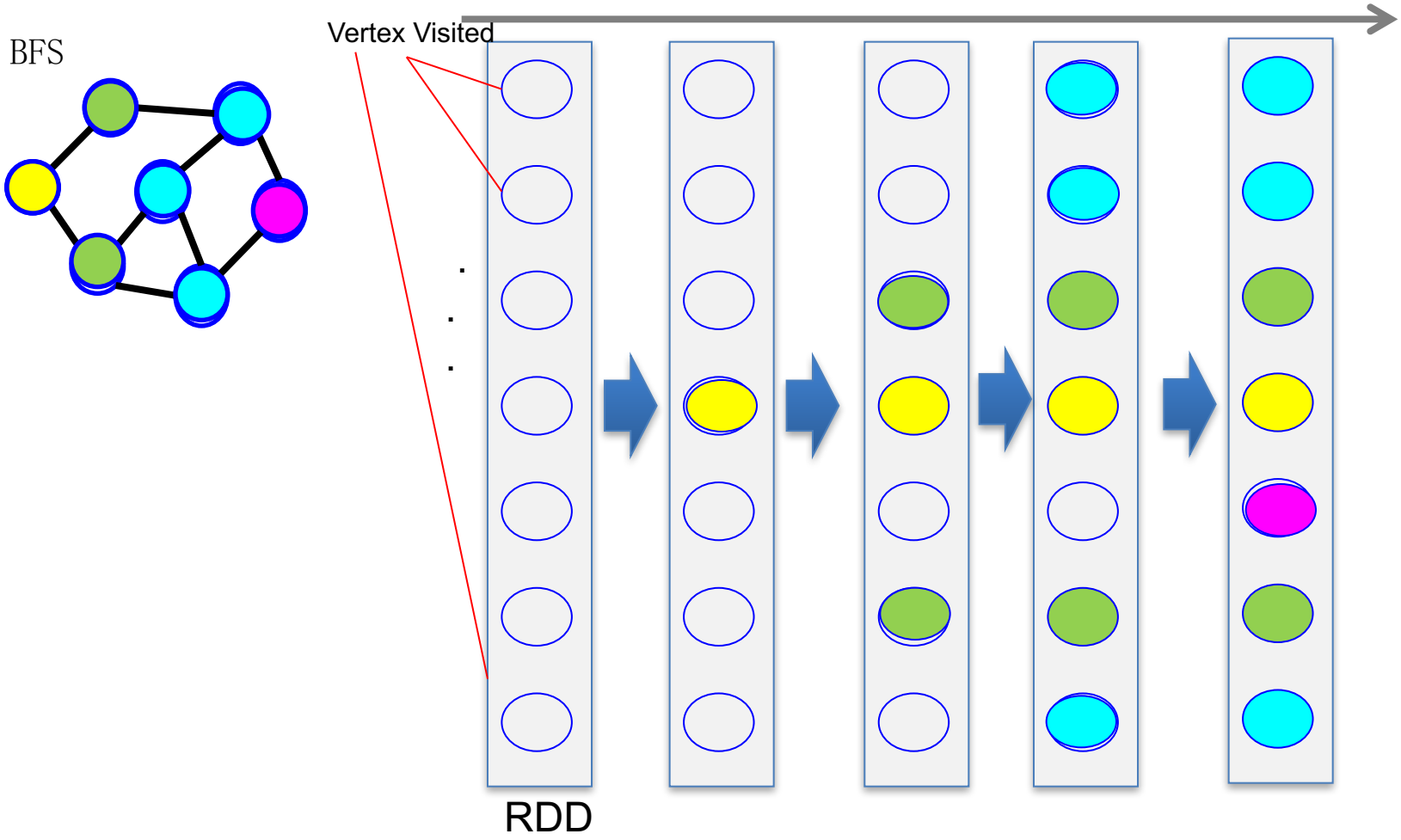
Breath-first-search

```
function breadth-first-search(vertices, source)
    frontier ← {source}
    next ← {}
    parents ← [-1,-1,...-1]
    while frontier ≠ {} do
        top-down-step(vertices, frontier, next, parents)
        frontier ← next
        next ← {}
    end while
    return tree
```

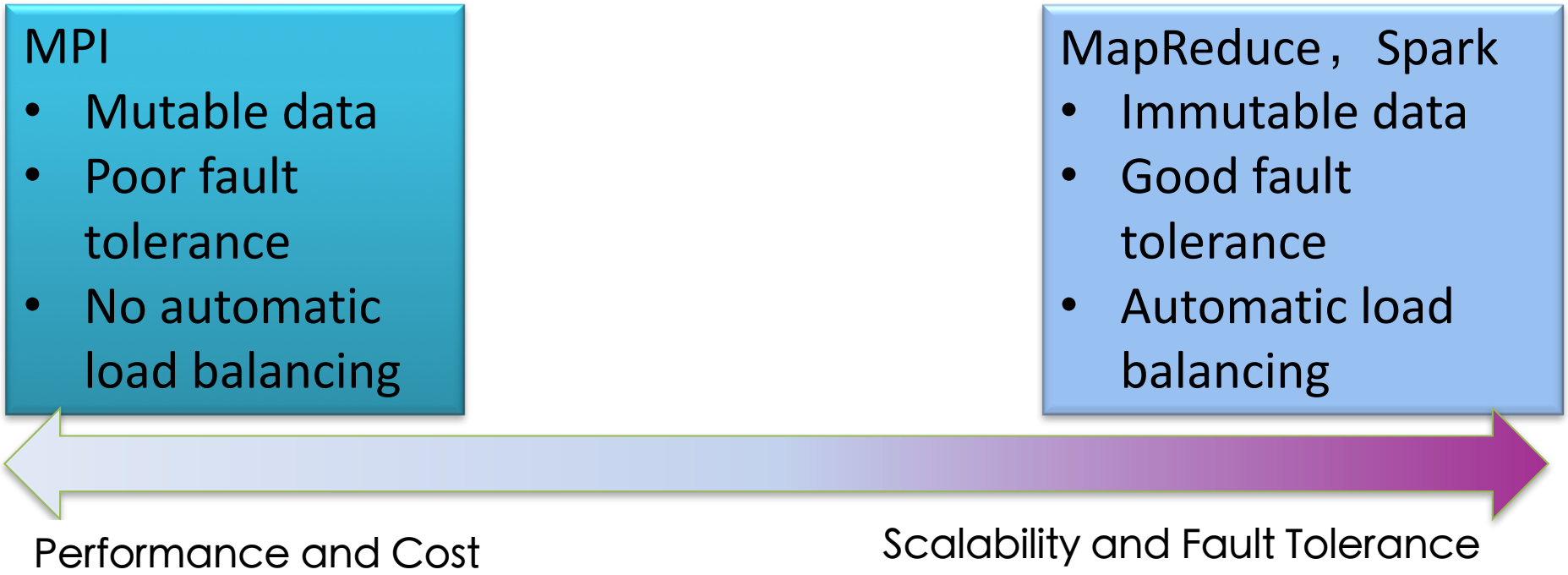
```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
```

Limitations of Spark

Spark： RDD



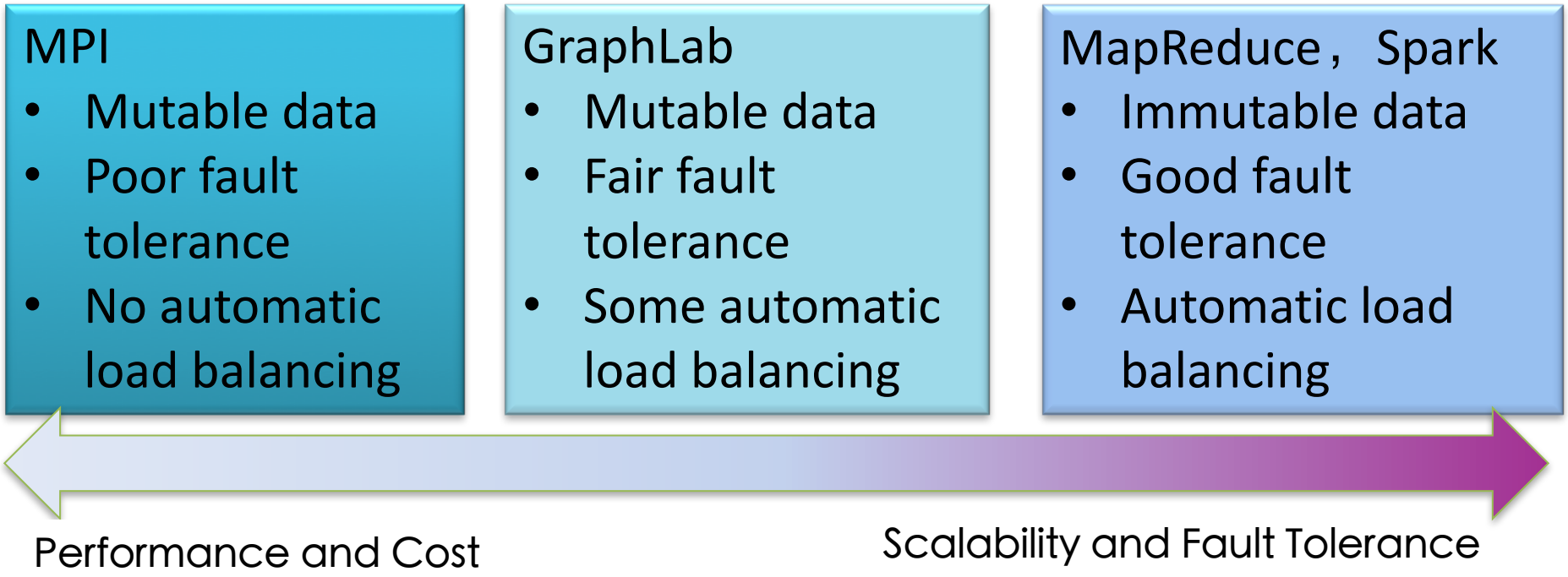
Trade-offs on the design of big data systems



Popular big data systems favor scalability and ease of programming than absolute performance

- Immutable data set is the key for good fault tolerance
- However, it damages the performance dramatically

Trade-offs on the design of big data systems



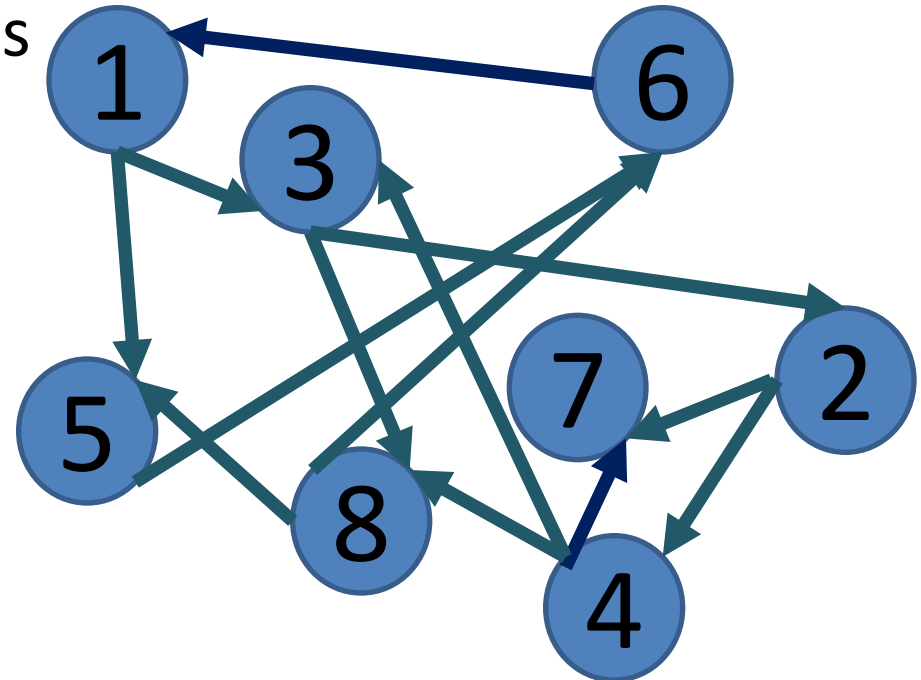
Graph Computing Model is something between MapReduce and MPI

Distributed Graph Computing Computing Model

Pregel and GraphLab1

Programming Abstraction for Graph

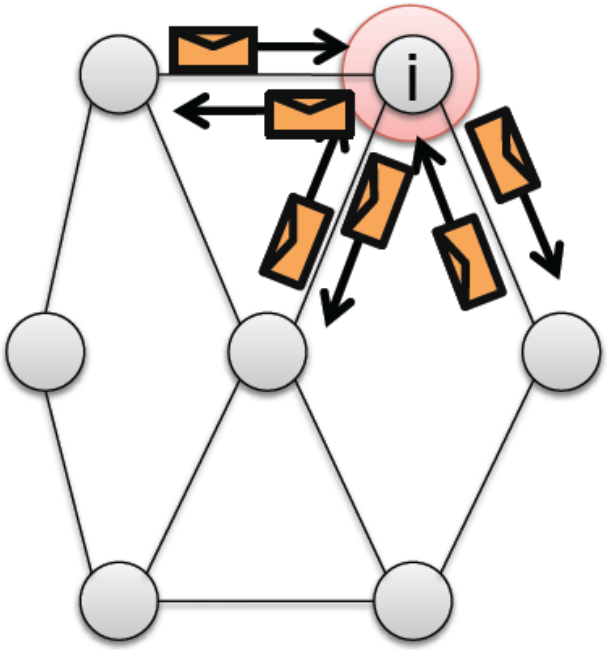
- Intuitive Programming abstractions
  - Vertices with state, computing change the state of vertices
    - Mutable data objects
  - Communication via edges



The Pregel Abstraction

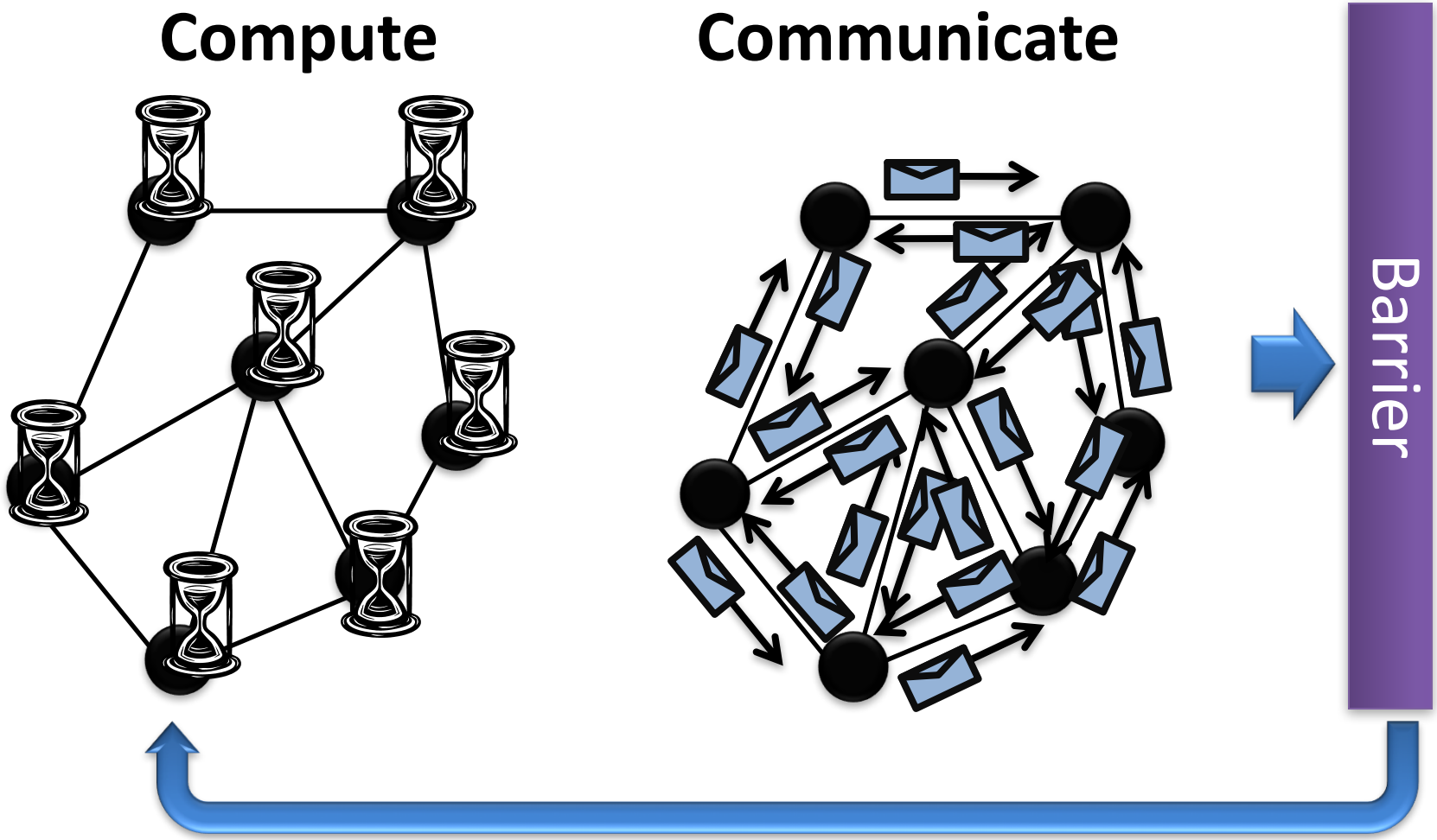
Vertex-Programs interact by sending **messages**.

```
Pregel_PageRank(i, messages) :
  // Receive all the messages
  total = 0
  foreach( msg in messages) :
    total = total + msg

  // Update the rank of this vertex
  R[i] = 0.15 + total

  // Send new messages to neighbors
  foreach(j in out_neighbors[i]) :
    Send  msg(R[i] * w_ij) to vertex j
```



Bulk Synchronous Parallel Model: Pregel (Giraph)

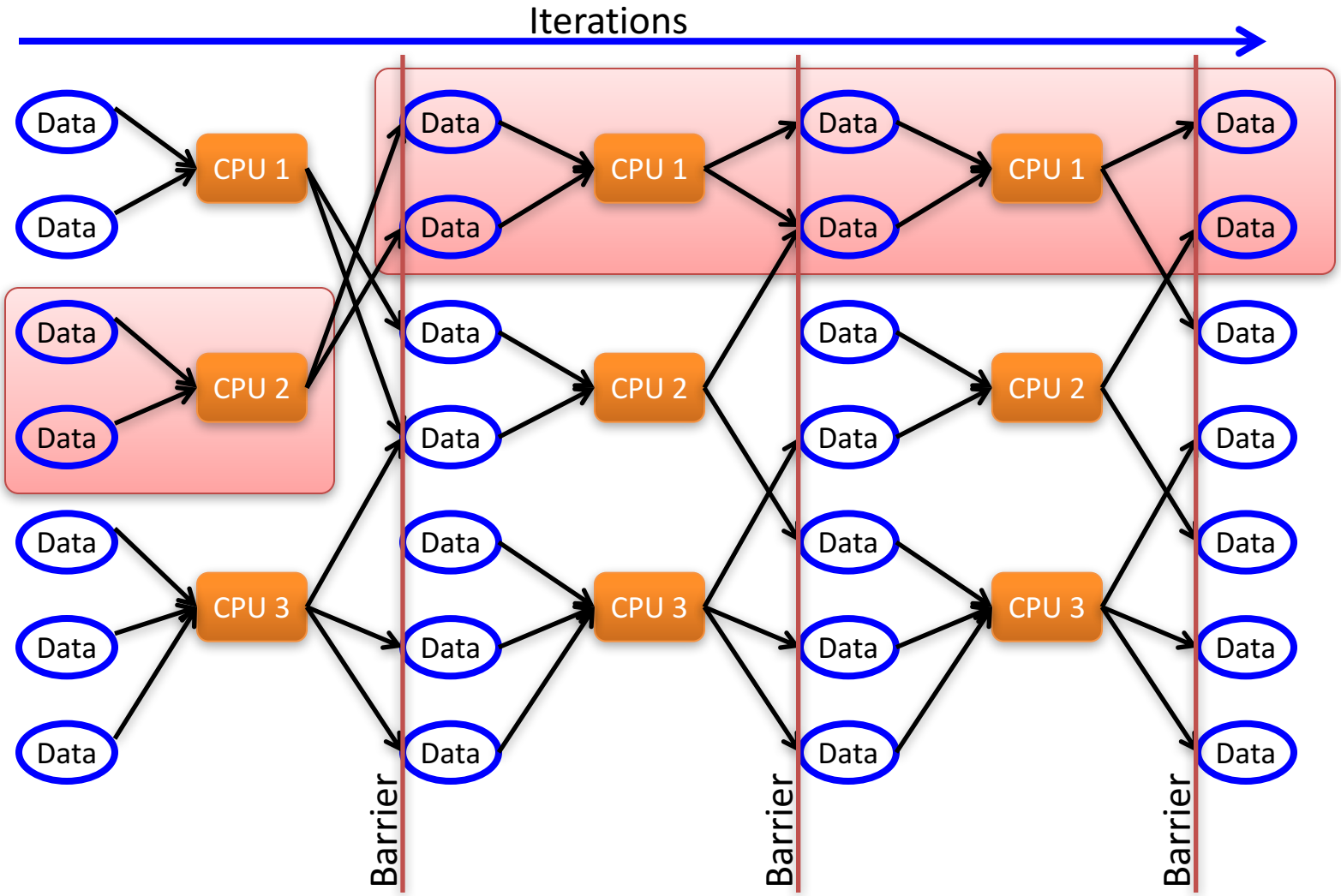
[Valiant '90]



What are the pros and cons of BSP +Mutable data objects?

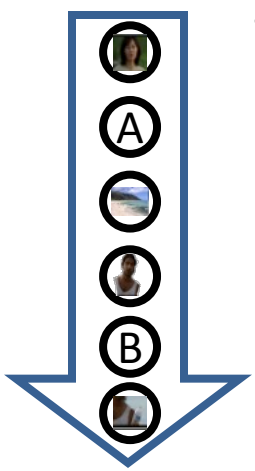
- Fault tolerance
  - Lineage is not possible
  - Enable un-coordinated checkpoints
- Performance
  - No unnecessary data object creation and memory copy
  - Synchronous model suffers from load imbalance

BSP Systems Problem: Curse of the Slow Job

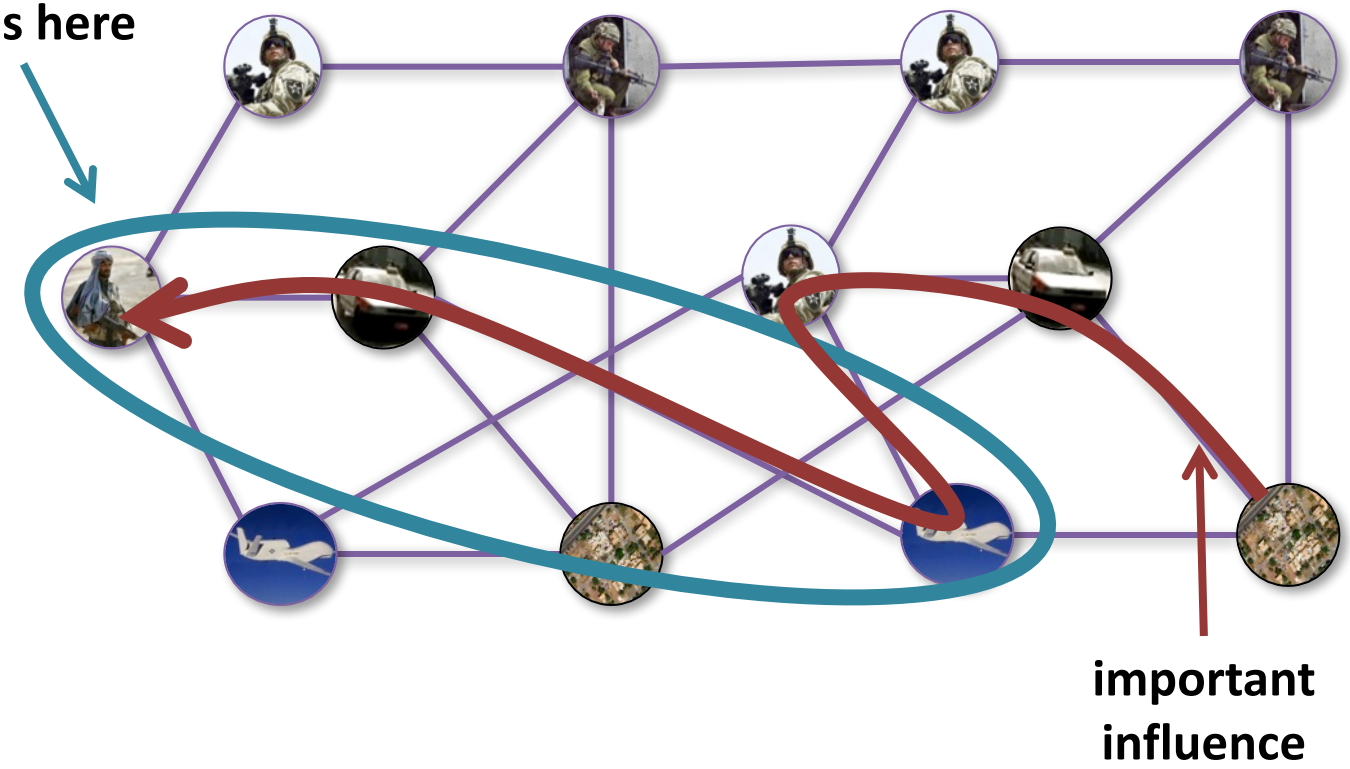


Analyzing Belief Propagation

[Gonzalez, Low, G. '09]



Priority Queue
Smart Scheduling

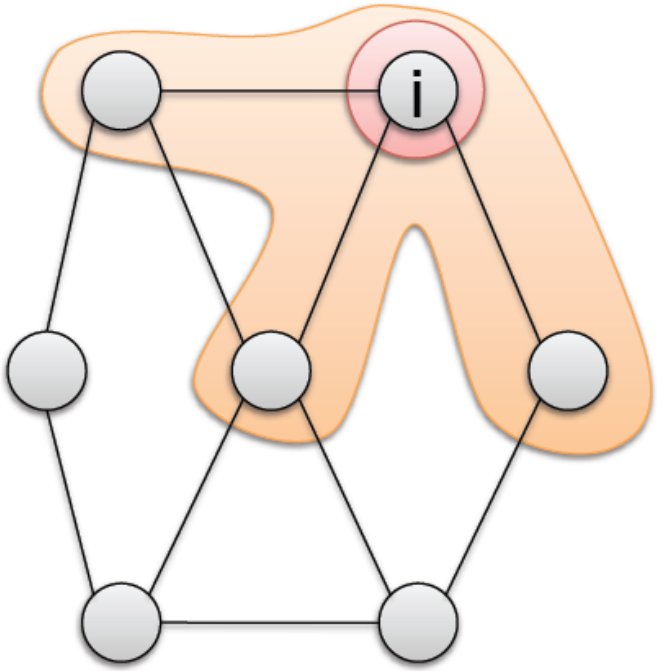


Asynchronous Parallel Model (rather than BSP) fundamental for efficiency

The GraphLab Abstraction

Vertex-Programs directly **read** the neighbors state

```
GraphLab_PageRank(i)
  // Compute sum over neighbors
  total = 0
  foreach( j in in_neighbors(i)):
    total = total + R[j] * w_ji

  // Update the PageRank
  R[i] = 0.15 + total

  // Trigger neighbors to run again
  if R[i] not converged then
    foreach( j in out_neighbors(i)):
      signal vertex-program on j
```



The Need for a New Abstraction

- Need: Asynchronous, Dynamic Parallel Computations

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

BSP, e.g., Pregel

Carnegie Mellon

Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

Semi-Supervised Learning
Label Propagation
CoEM

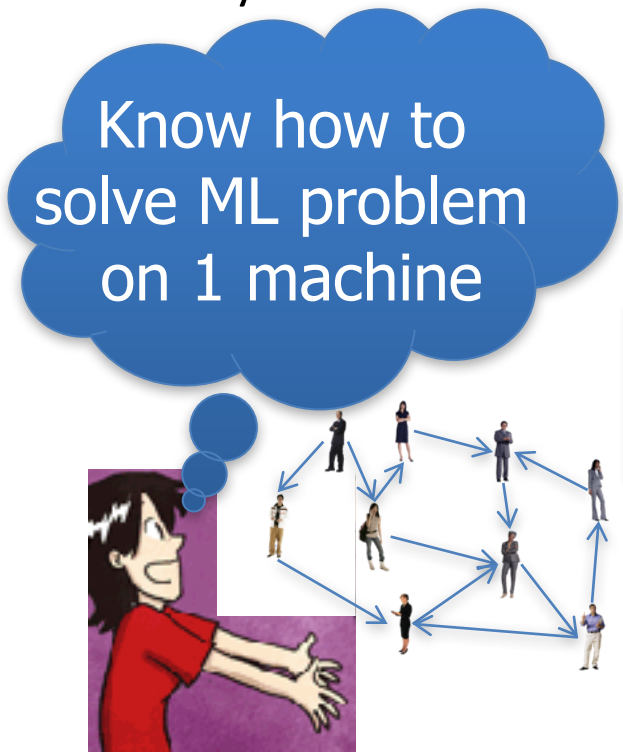
Collaborative Filtering
Tensor Factorization

Data-Mining
PageRank
Triangle Counting

The **GraphLab** Goals

- Designed specifically for ML
  - Graph dependencies
  - Iterative
  - Asynchronous
  - Dynamic

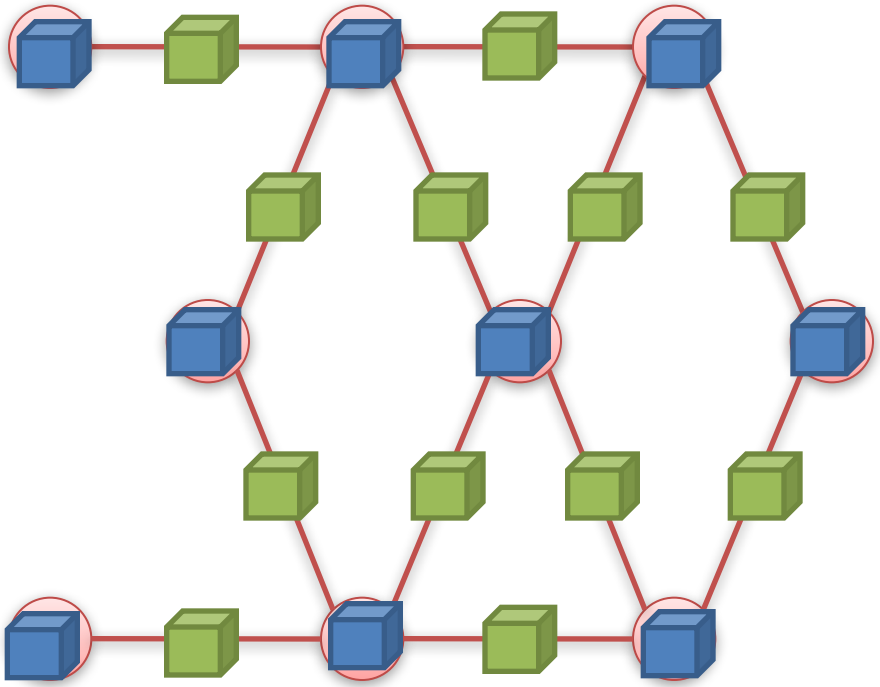
- Simplifies design of parallel programs:
  - Abstract away hardware issues
  - Automatic data synchronization
  - Addresses multiple hardware architectures



Efficient parallel predictions

Data Graph

Data associated with vertices and edges



Graph:

- Social Network

Vertex Data:

- User profile text
- Current interests estimates

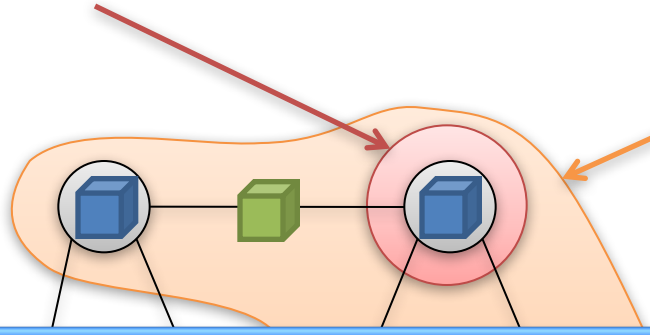
Edge Data:

- Similarity weights

Update Functions

User-defined program: applied to

vertex transforms data in **scope** of vertex



```
pagerank(i, scope){
  // Get Neighborhood data
  (R[i], w_ij, R[j]) ←scope;
```

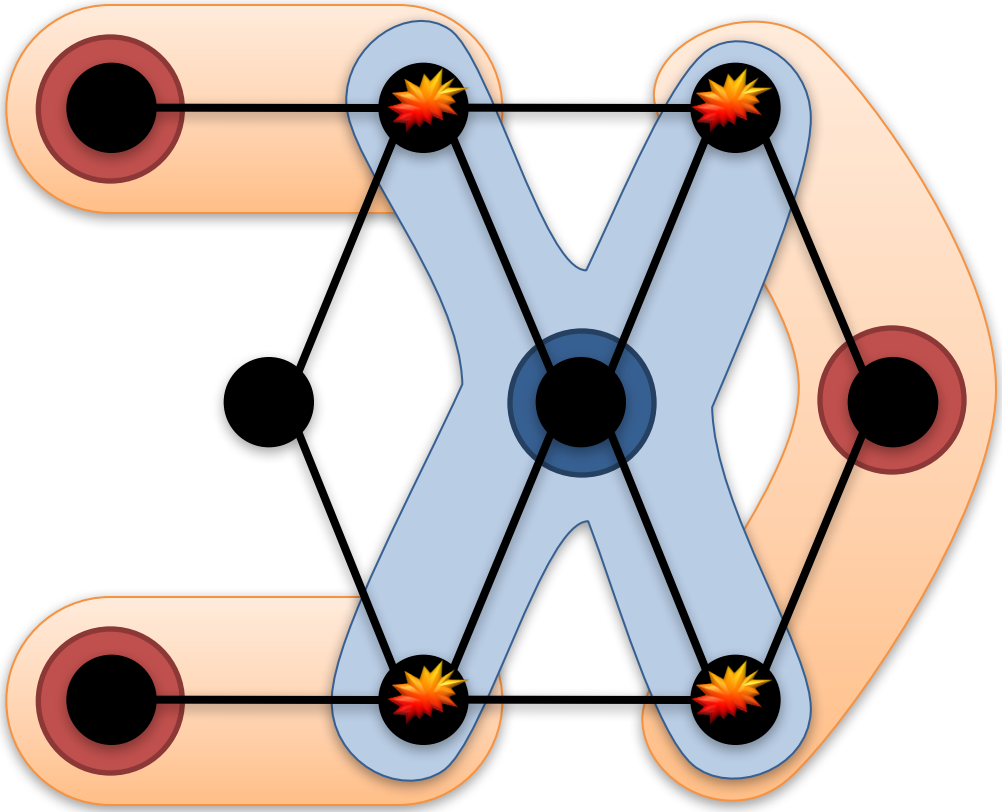
Update function applied (asynchronously) in parallel until convergence

Many schedulers available to prioritize computation

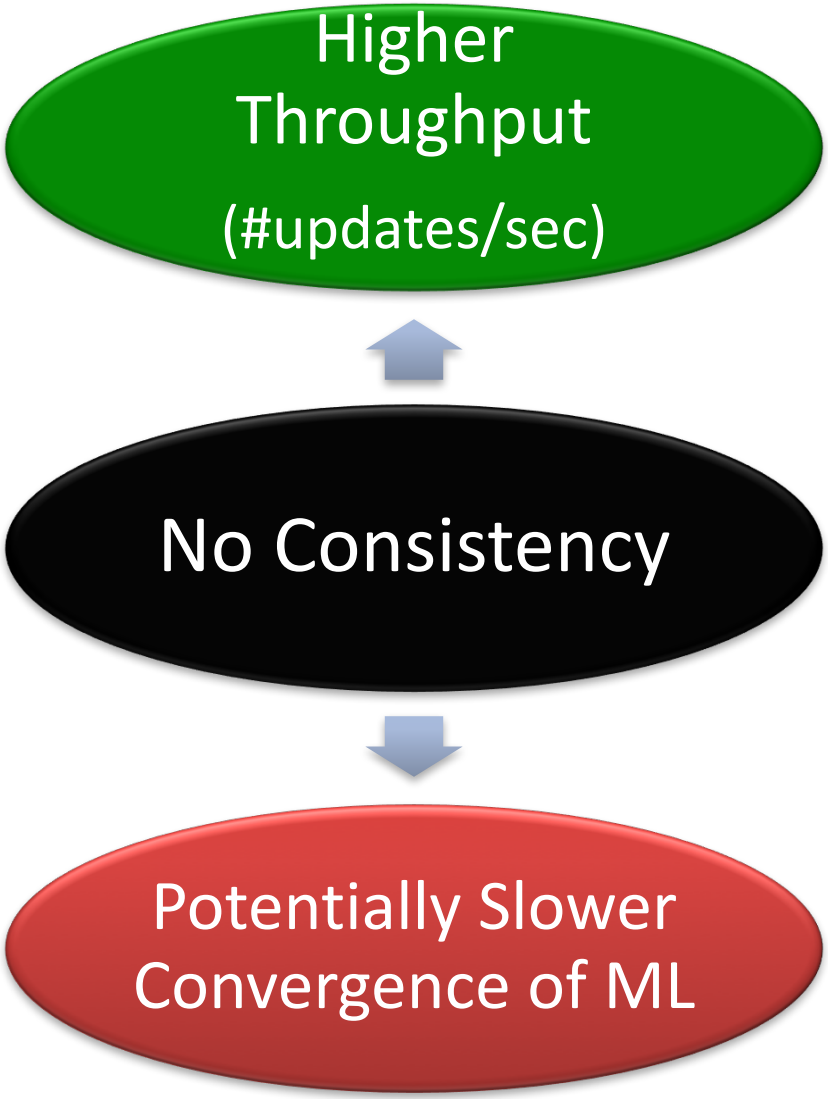
Dynamic computation

Ensuring Race-Free Code

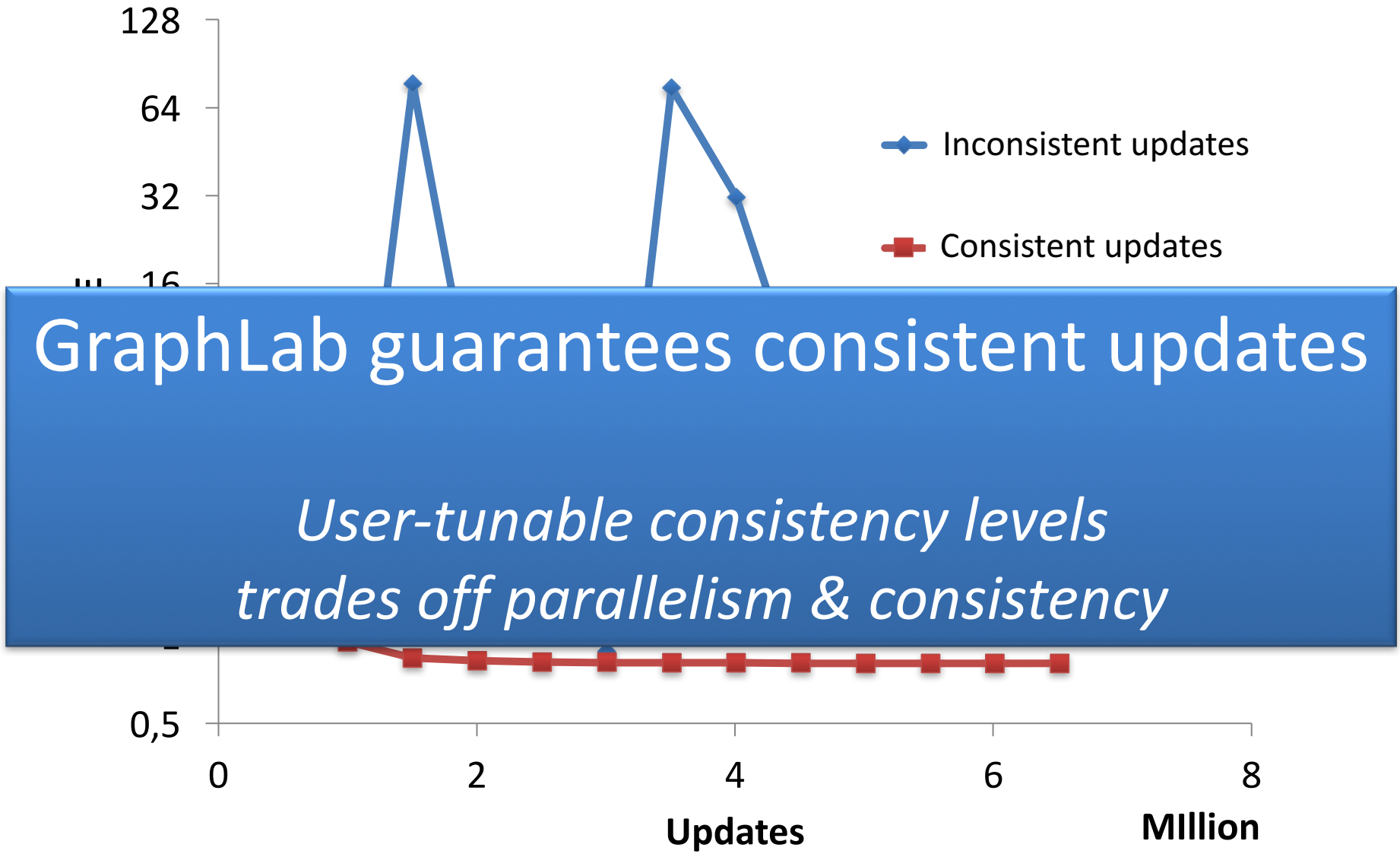
How much can computation **overlap**?



Need for Consistency?



Consistency in Collaborative Filtering



GraphLab guarantees consistent updates

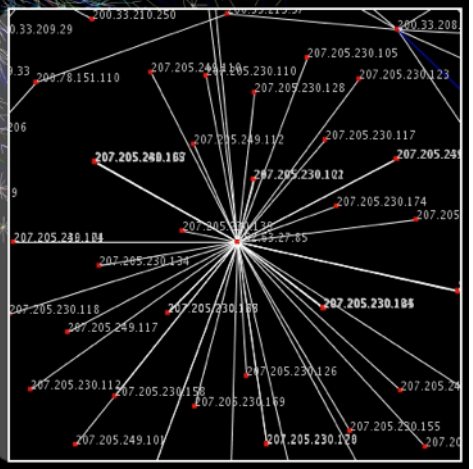
User-tunable consistency levels
trades off parallelism & consistency

Netflix data, 8 cores

Natural Graph Partitioning

GraphLab 2

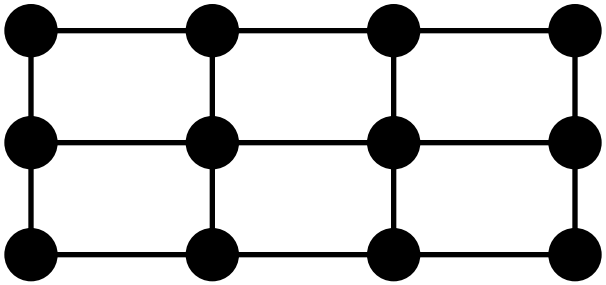
Natural Graphs



ons]

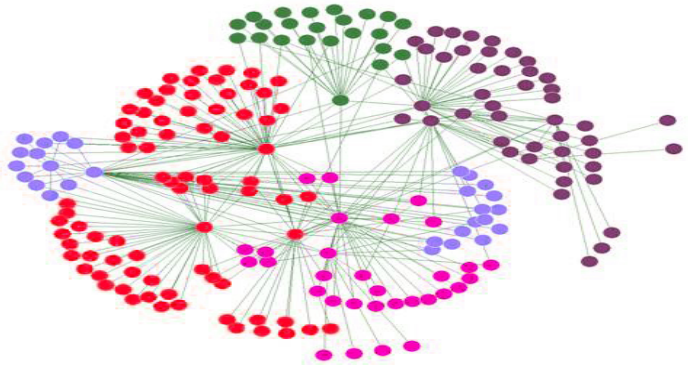
Assumptions of Graph-Parallel Abstractions

Idealized Structure



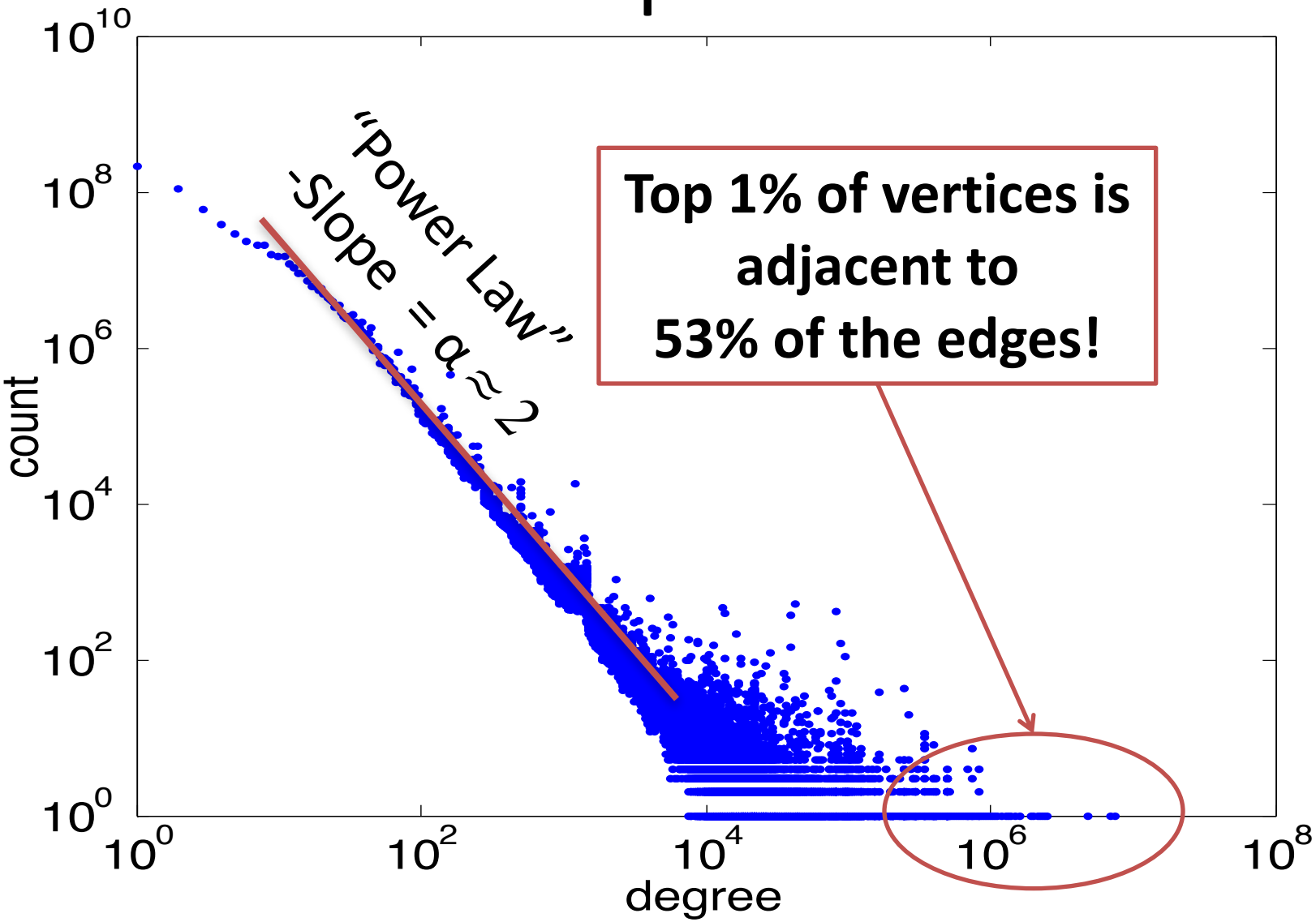
- ***Small*** neighborhoods
  - Low degree vertices
- Vertices have similar degree
- Easy to partition

Natural Graph



- ***Large*** Neighborhoods
  - High degree vertices
- ***Power-Law*** degree distribution
- ***Difficult to partition***

Natural Graphs → Power Law

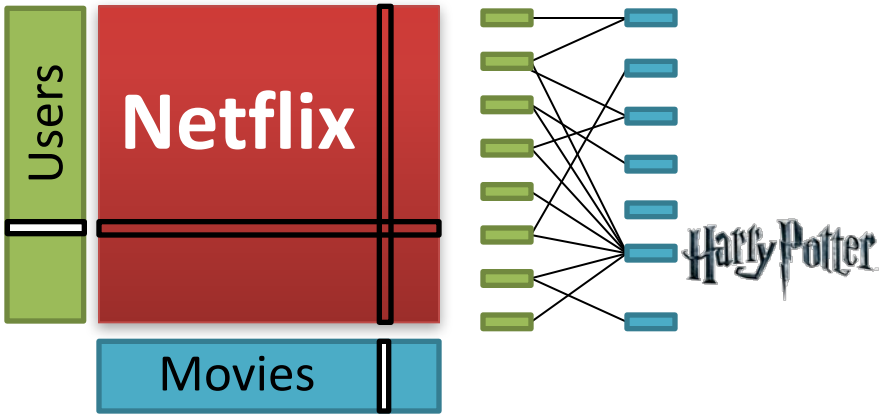


Altavista Web Graph: 1.4B Vertices, 6.7B Edges

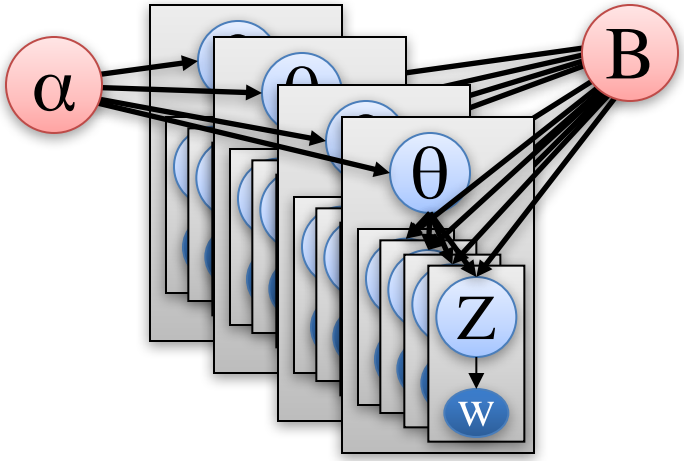
High Degree Vertices are Common

"Social" People

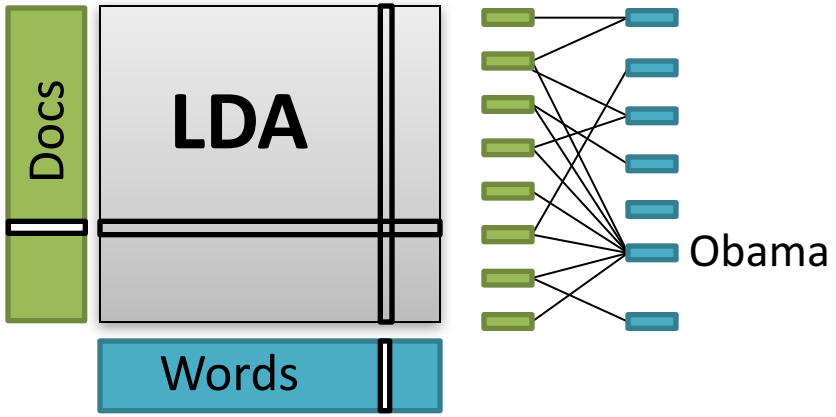
Popular Movies



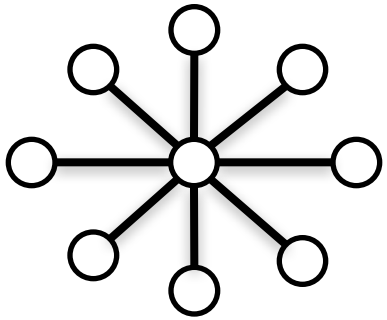
Hyper Parameters



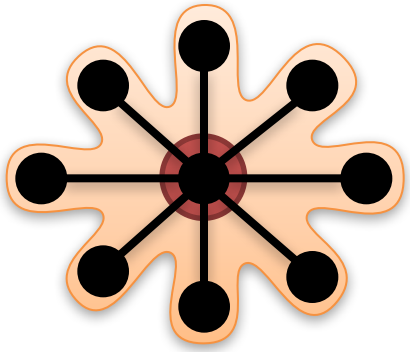
Common Words



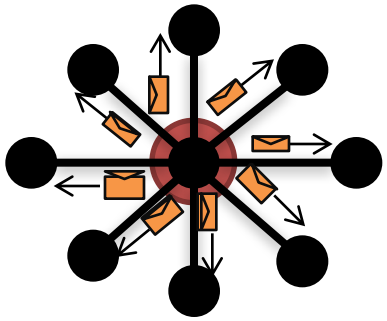
Problem:
High Degree Vertices Limit Parallelism



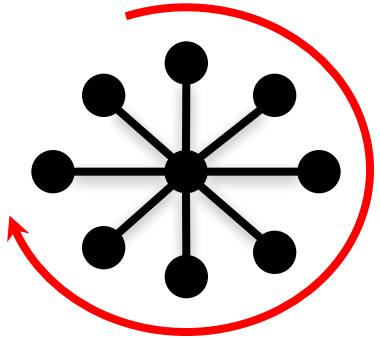
Edge information too large for single machine



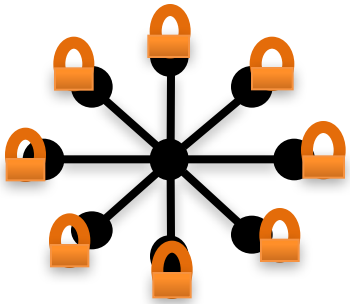
Touches a large fraction of graph (GraphLab 1)



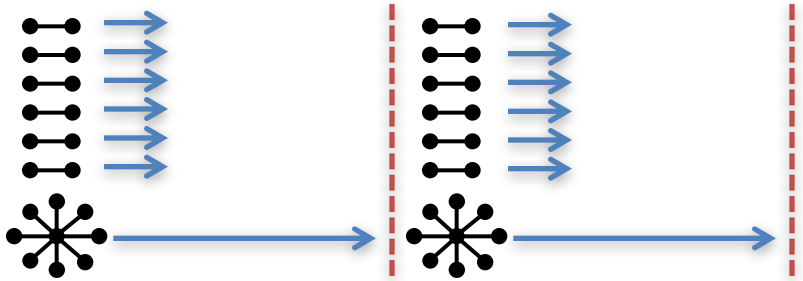
Produces many messages (Pregel)



Sequential Vertex-Updates



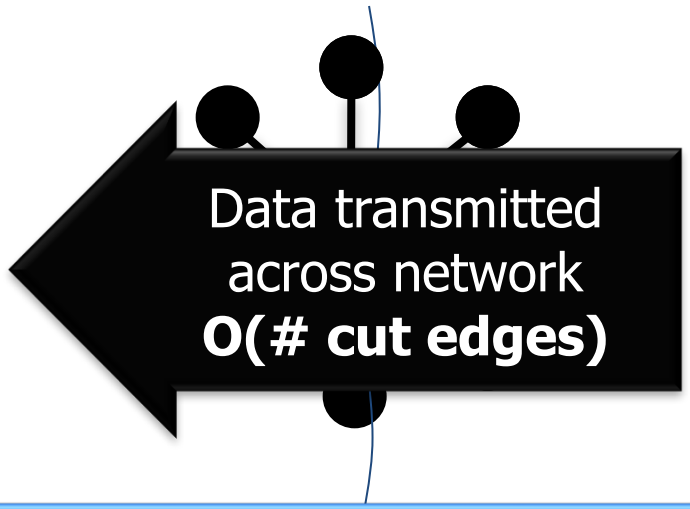
Asynchronous consistency requires heavy locking (GraphLab 1)



Synchronous consistency is prone to stragglers (Pregel)

Problem:

High Degree Vertices ➔ High Communication for Distributed Updates



Natural graphs do not have low-cost balanced cuts

[Leskovec et al. 08, Lang 04]

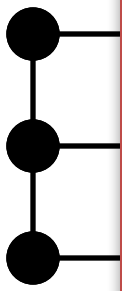
Popular partitioning tools (Metis, Chaco,…) perform poorly

[Abou-Rjeili et al. 06]

Extremely slow and require substantial memory

Random Partitioning

- Both GraphLab 1 and Pregel proposed Random (hashed) partitioning for Natural Graphs



For *p* Machines:

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|E|}\right] = 1 - \frac{1}{p}$$

10 Machines → 90% of edges cut

100 Machines → 99% of edges cut!

In Summary

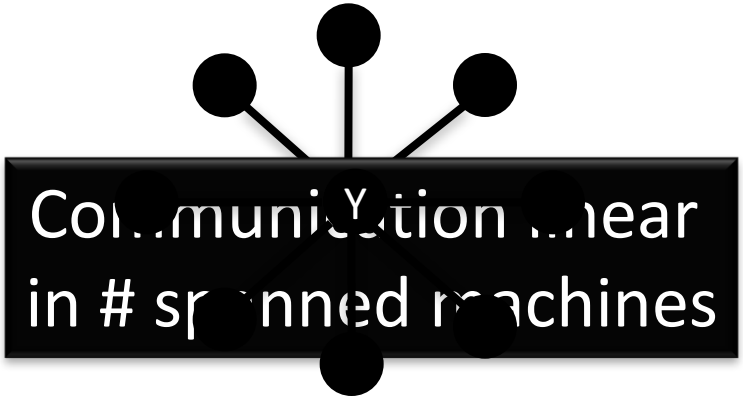
GraphLab 1 and Pregel are not well suited for natural graphs

- Poor performance on high-degree vertices
- Low Quality Partitioning

- Distribute a single vertex-update
  - Move computation to data
  - Parallelize high-degree vertices

- Vertex Partitioning
  - Simple online approach, effectively partitions large power-law graphs

Minimizing Communication in GraphLab 2: **Vertex Cuts**



A **vertex-cut** minimizes
machines per vertex

Percolation theory suggests Power Law graphs can be split by removing only a small set of vertices [Albert et al. 2000]

➔

Small vertex cuts possible!

A **Common Pattern** for Vertex-Programs

GraphLab_PageRank(i)

```
// Compute sum over neighbors
total = 0
foreach( j in in_neighbors(i)):
  total = total + R[j] * w_ji
```

Gather Information About Neighborhood

```
// Update the PageRank
R[i] = 0.1 + total
```

Update Vertex

```
// Trigger neighbors to run again
if R[i] not converged then
  foreach( j in out_neighbors(i))
    signal vertex-program on j
```

Signal Neighbors & Modify Edge Data

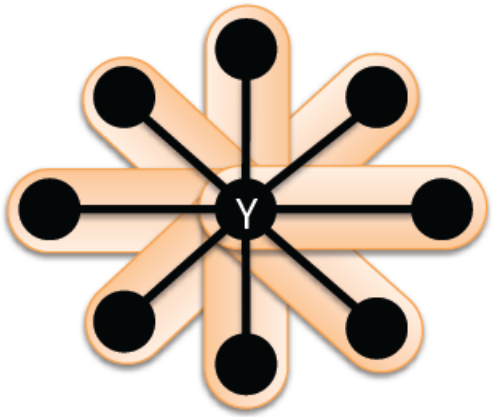
GAS Decomposition

Gather (Reduce)

Accumulate information about neighborhood

User Defined:

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



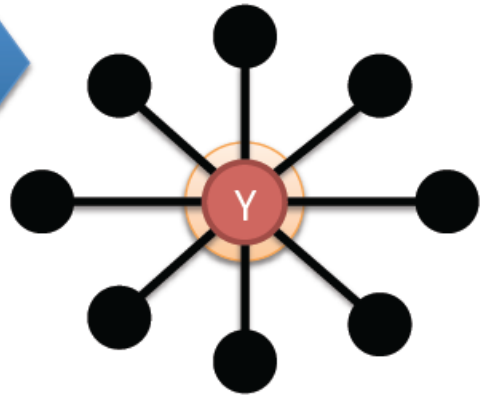
Parallel Sum  → Σ

Apply

Apply the accumulated value to center vertex

User Defined:

- Apply( , Σ) → 

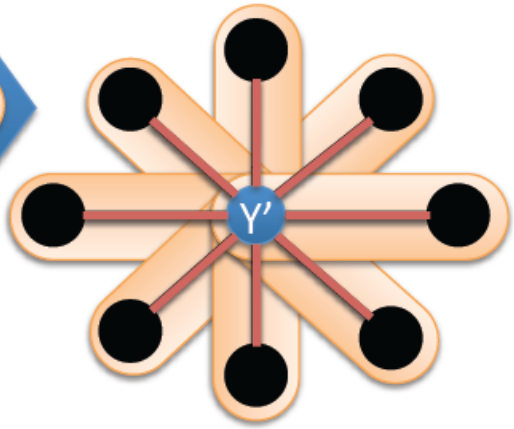


Scatter

Update adjacent edges and vertices.

User Defined:

- Scatter() → —



Update Edge Data & Activate Neighbors

PageRank in PowerGraph

$$R[i] = 0.15 + \sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]$$

PowerGraph_PageRank(i)

Gather(j → i) : return $w_{ji} * R[j]$

sum(a, b) : return a + b;

Apply(i, Σ) : $R[i] = 0.15 + \Sigma$

Scatter(i → j) :

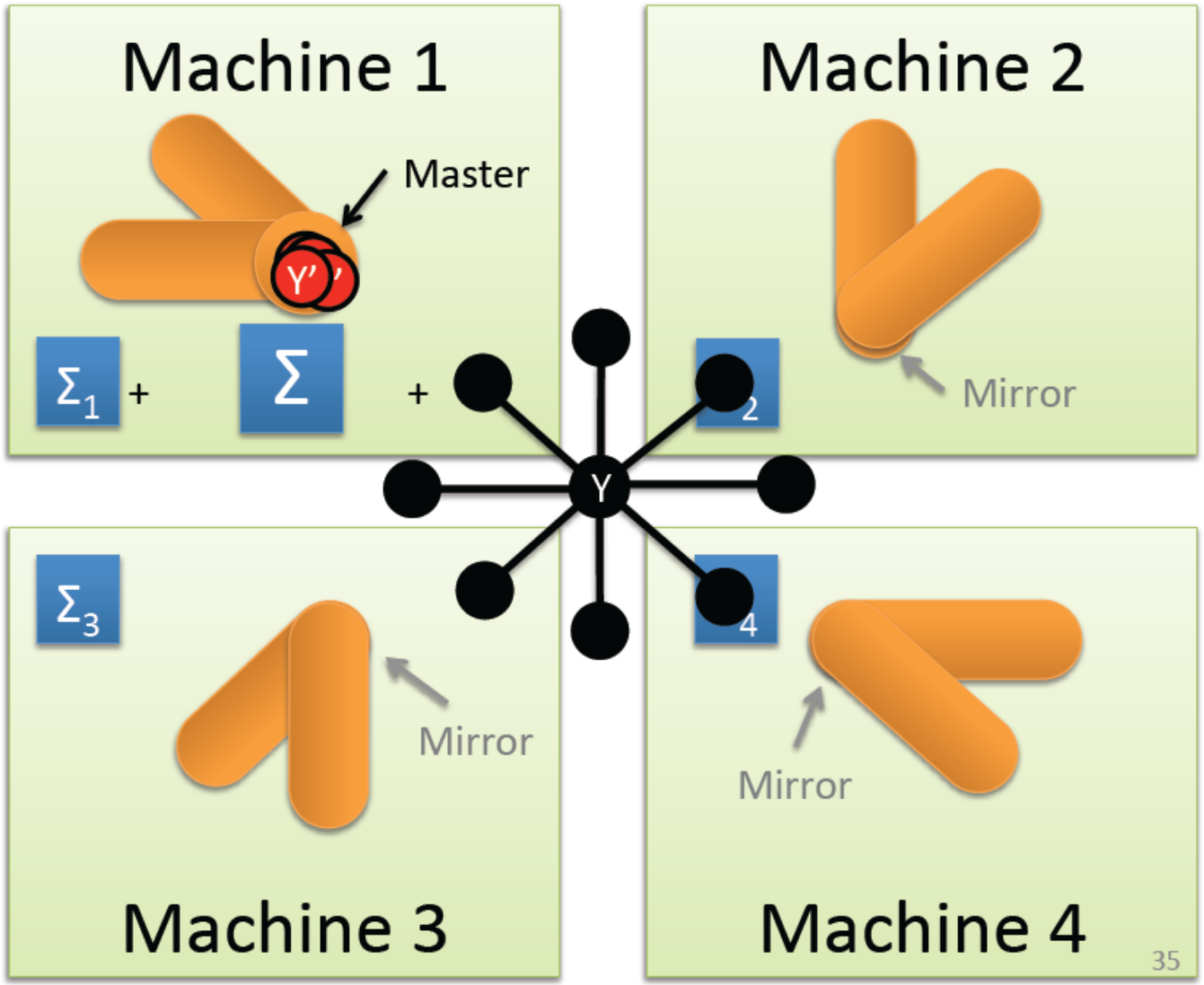
if *R[i]* changed then trigger *j* to be **recomputed**

Distributed Execution of a PowerGraph Vertex-Program

Gather

Apply

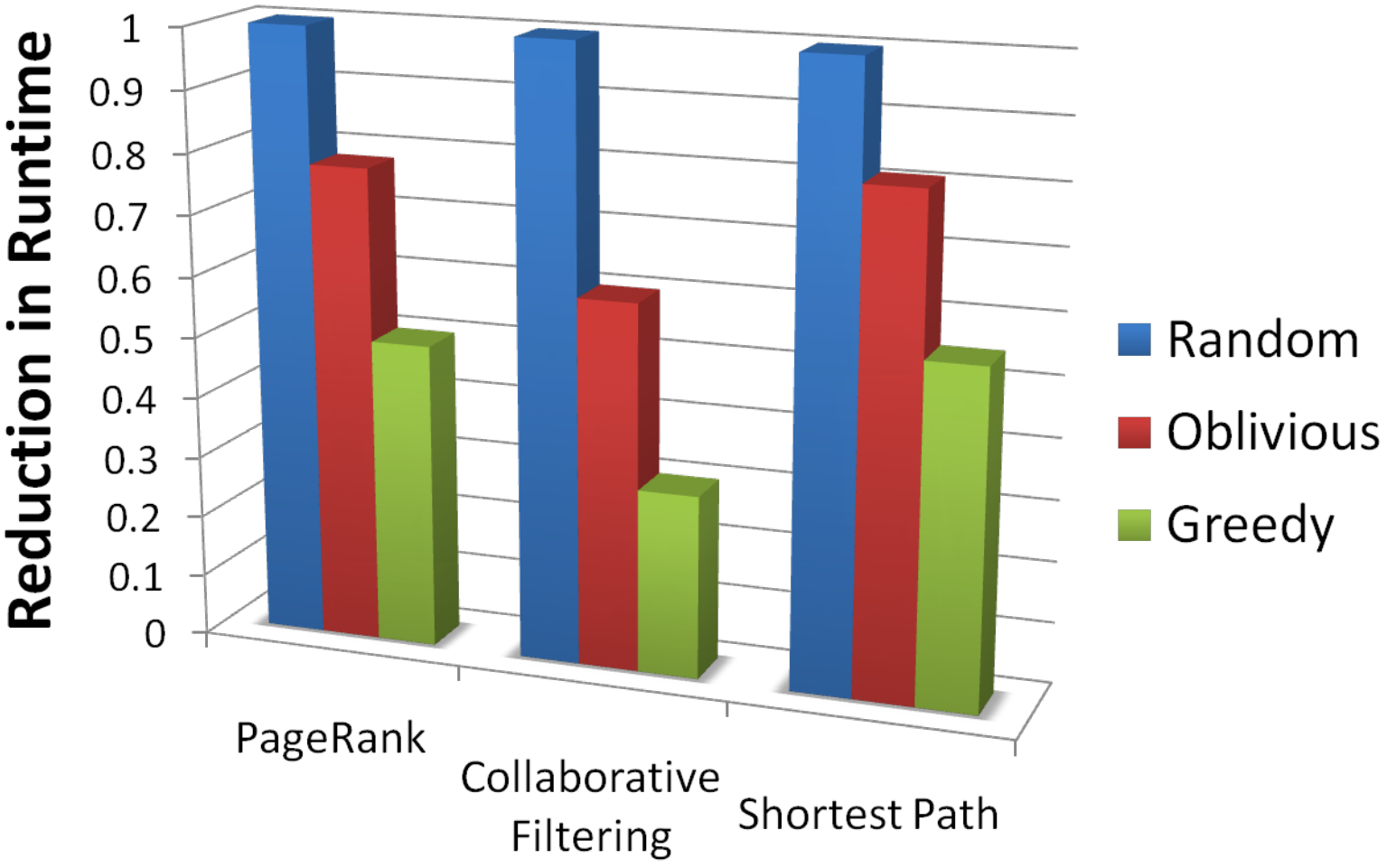
Scatter



Constructing Vertex-Cuts

- **Goal:** *Parallel graph partitioning on ingress*
- GraphLab 2 provides three **simple** approaches:
  - ***Random*** *Edge Placement*
    - Edges are placed randomly by each machine
      - Good theoretical guarantees
  - ***Greedy*** *Edge Placement with* ***Coordination***
    - Edges are placed using a shared objective
      - Better theoretical guarantees
  - ***Oblivious-Greedy*** *Edge Placement*
    - Edges are placed using a local objective

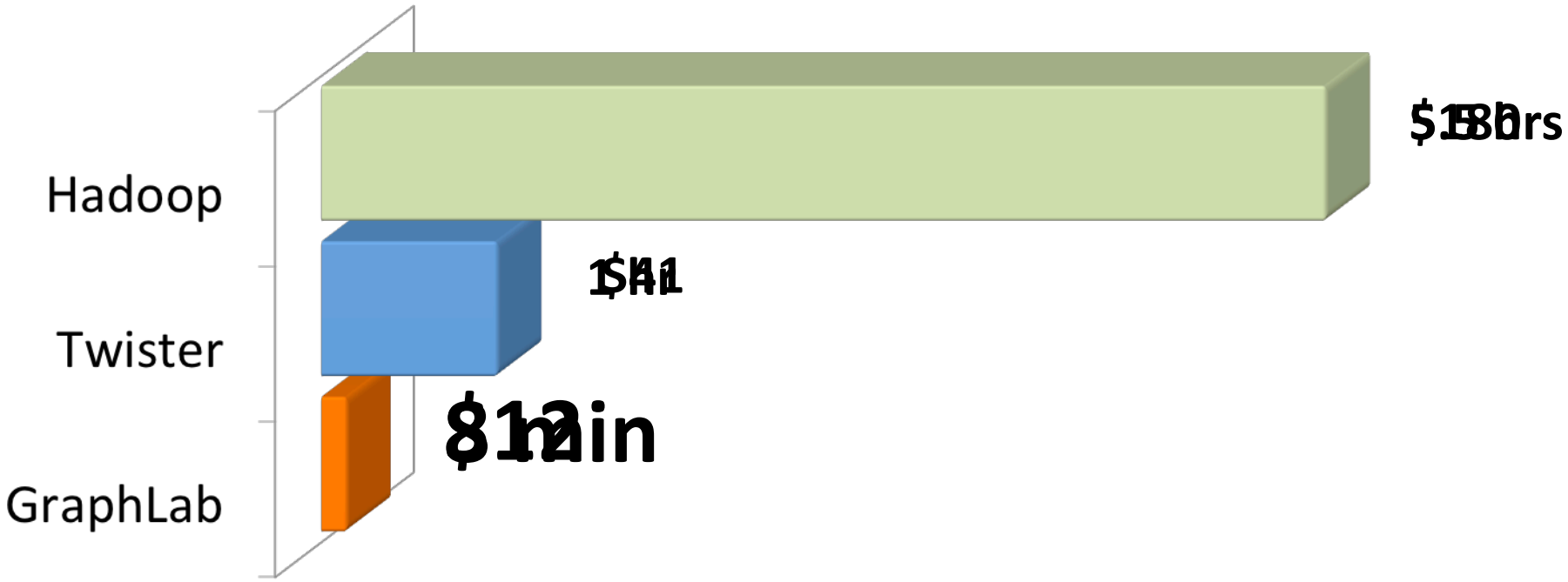
Beyond Random Vertex Cuts!



LDA Performance

- All English language Wikipedia
  - 2.6M documents, 8.3M words, 500M tokens
- LDA state-of-the-art sampler (100 Machines)
  - *Alex Smola*: 150 Million tokens per Second
- GraphLab Sampler (64 cc2.8xlarge EC2 Nodes)
  - **100 Million Tokens per Second**
  - Using only 200 Lines of code and 4 human hours

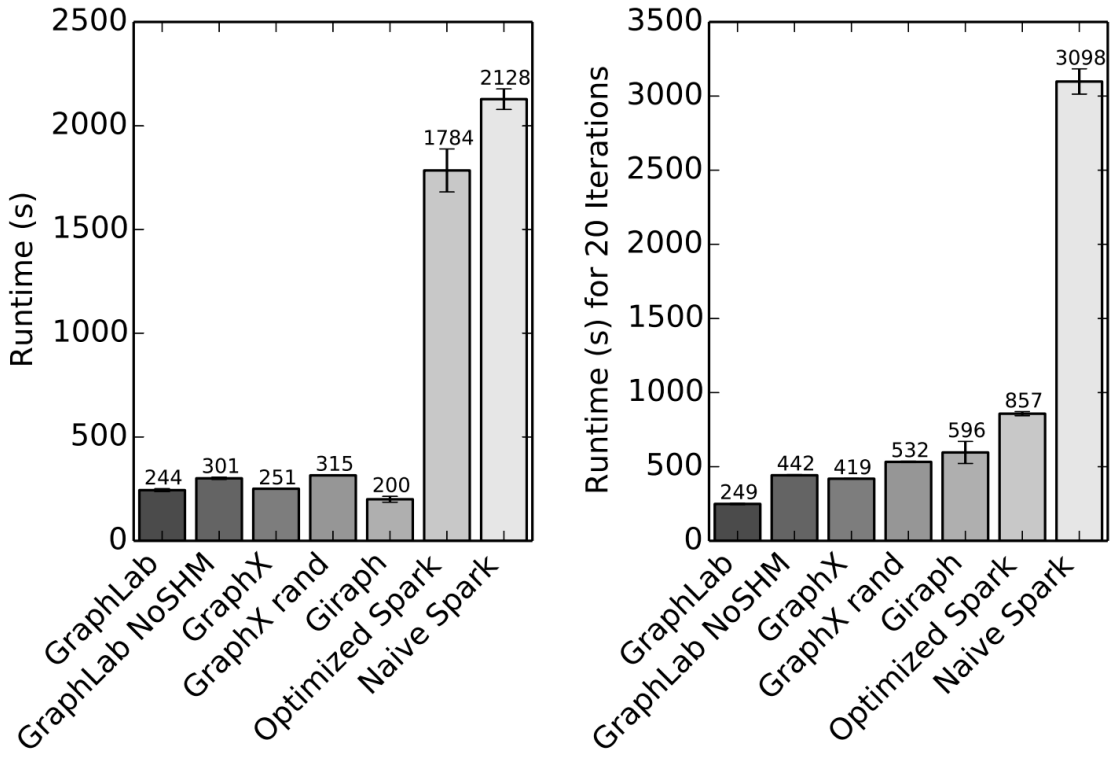
PageRank



40M Webpages, 1.4 Billion Links

Hadoop results from [Kang et al. '11]
Twister (in-memory MapReduce) [Ekanayake et al. '10]

GraphLab is 10 times faster than Spark for some tasks



(a) Conn. Comp. Twitter (b) PageRank Twitter

Gonzalez, Joseph E., et al. "Graphx: Graph processing in a distributed dataflow framework." Proceedings of OSDI. 2014.

But GraphLab still has its issues

- Memory Usage
  - 10X memory size required
- Poor locality and load balance
  - GraphLab on 8 machines is slower than single node system on 1 machine

System	PageRank	ConnComp
Ligra [29]	44.1	7.46
Galois [21]	19.0	11.5
Polymer [34]	43.1	7.14
PowerGraph [11]	40.3	29.1
GraphX [12]	216	104
PowerLyra [9]	26.9	22.0

A even faster Distributed Graph Engine

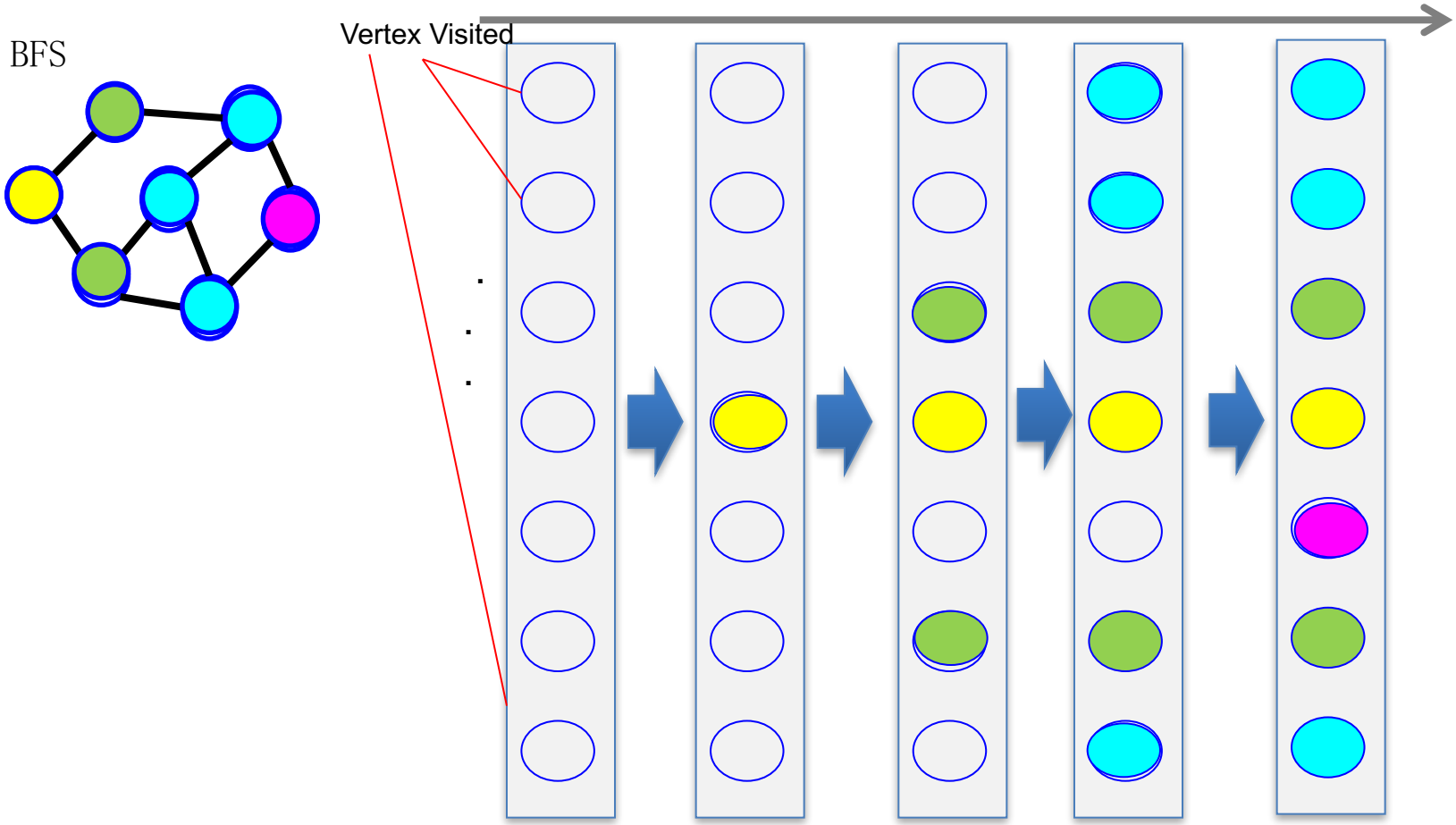
Gemini

Review: Breath-first-search

```
function breadth-first-search(vertices, source)
    frontier ← {source}
    next ← {}
    parents ← [-1,-1,...-1]
    while frontier ≠ {} do
        top-down-step(vertices, frontier, next, parents)
        frontier ← next
        next ← {}
    end while
    return tree
```

```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
```

Number of Active vertices



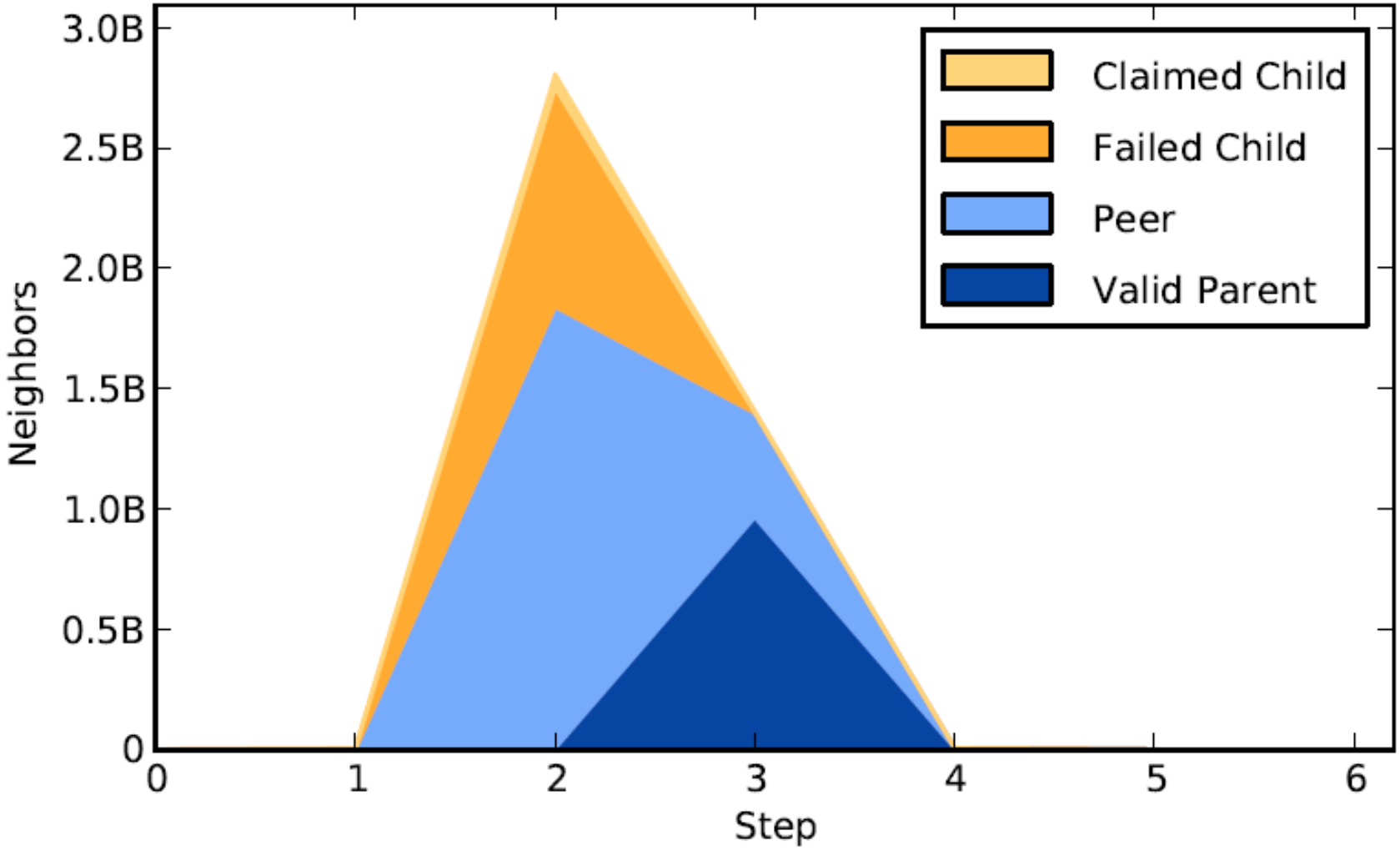
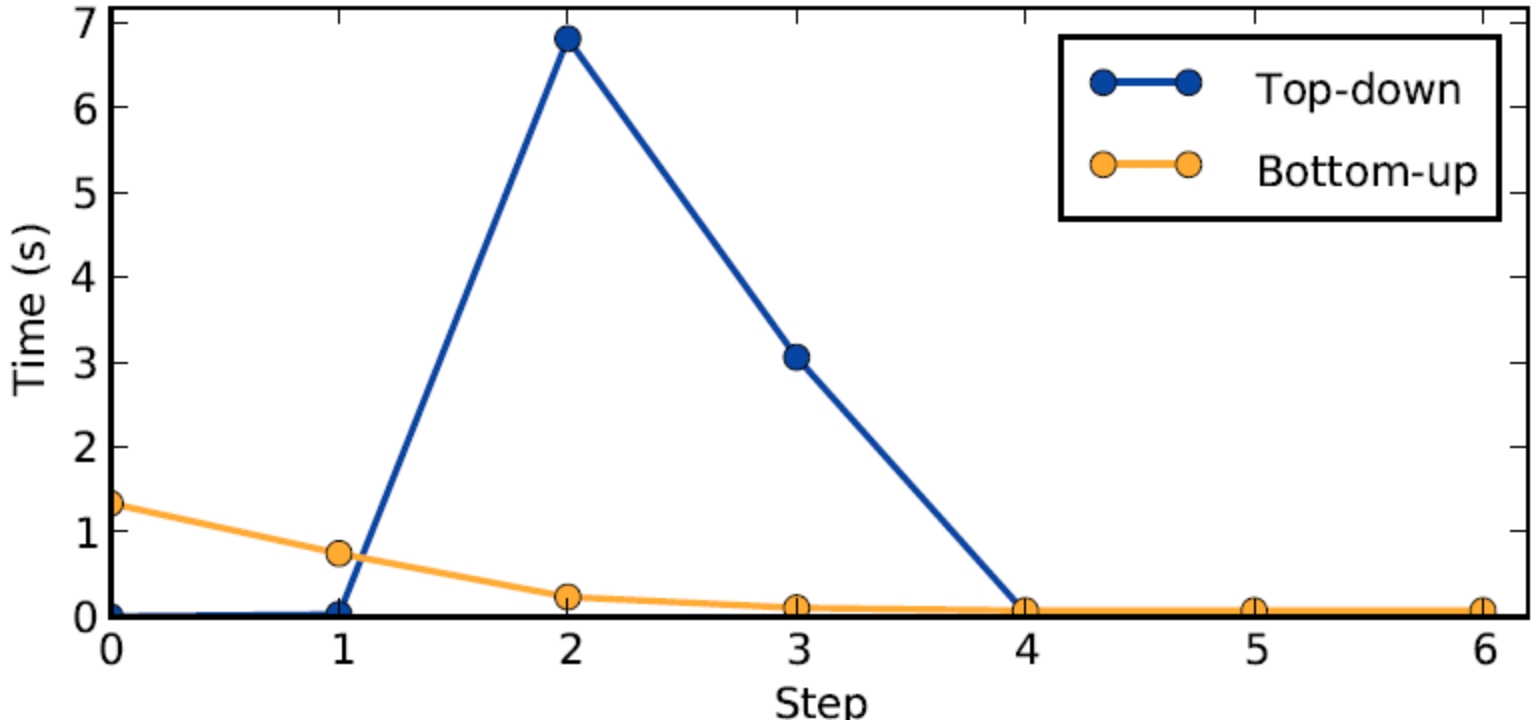


Fig. 3. Breakdown of edges in the frontier for a sample search on `kron27` (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system.

Botton-up BFS

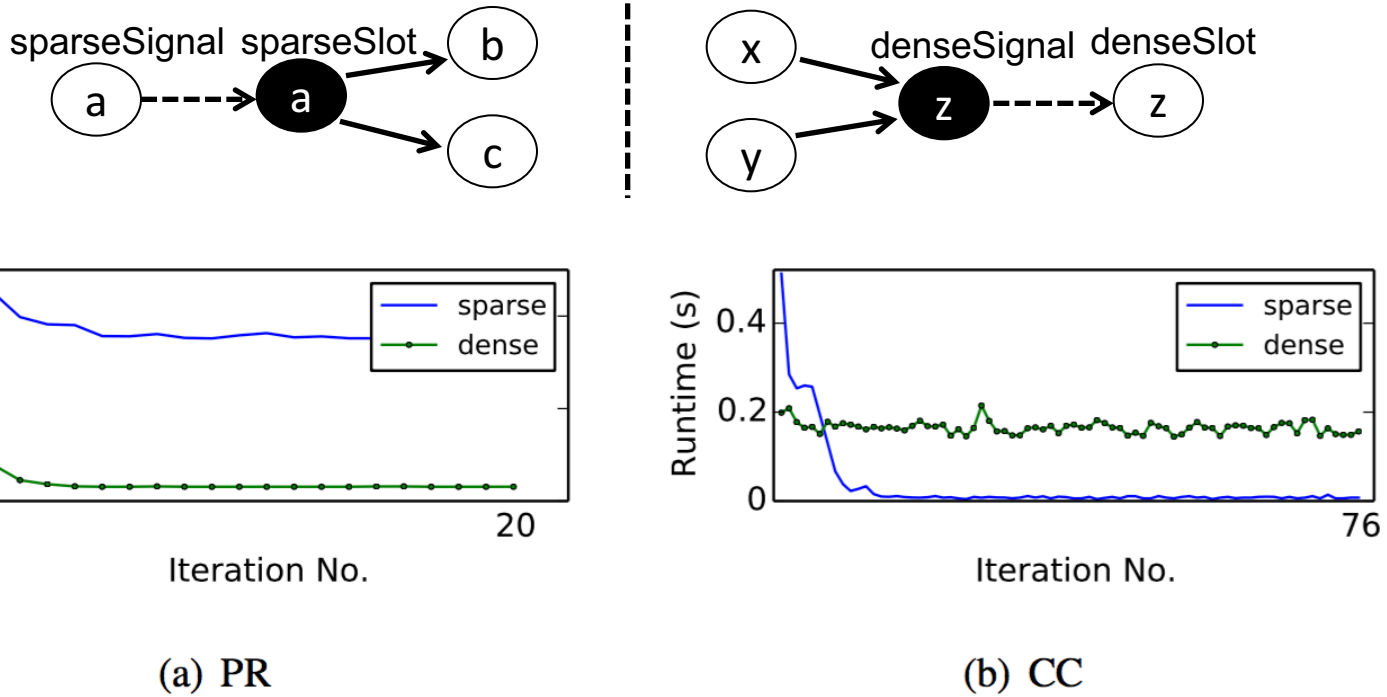
```
function bottom-up-step(vertices, frontier, next, parents)
    for v ∈ vertices do
        if parents[v] = -1 then
            for n ∈ neighbors[v] do
                if n ∈ frontier then
                    parents[v] ← n
                    next ← next ∪ {v}
                    break
                end if
            end for
        end if
    end for
```

Motivation for a hybrid algorithm



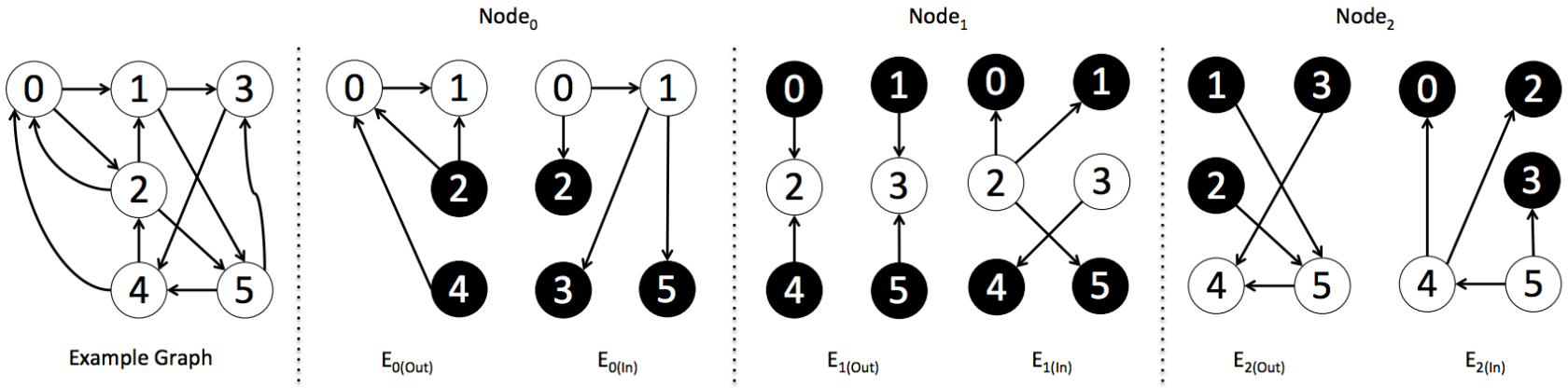
Gemini - Extend to Distributed Systems

- Sparse-dense dual engine on distributed systems
  - Number of active vertices changed overtime and favor different computing models

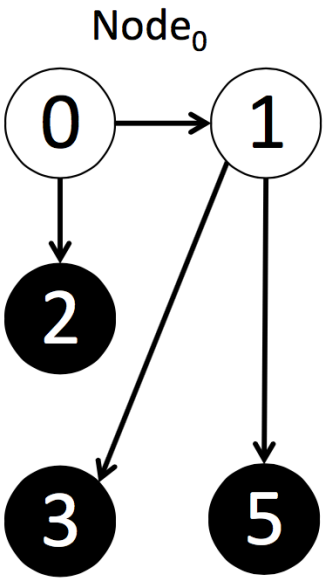


Chunk-based partition

- Previous approach
  - Hash-based : load balanced,but may lose locality
  - Min-Cut: Reduced communication, but too heavy
- Our approach: Chunk-based partition
  - Preserve locality, fast partition
  - Works well with additional scheduling techniques
  - Locality Sensitive Partitioning



Input Sensitive Edge Organization



CSC

nb = [0, 0, 1, 1]
idx = [0, 0, 1, 2, 3, 3, 4]
ext = 011101

} BAI

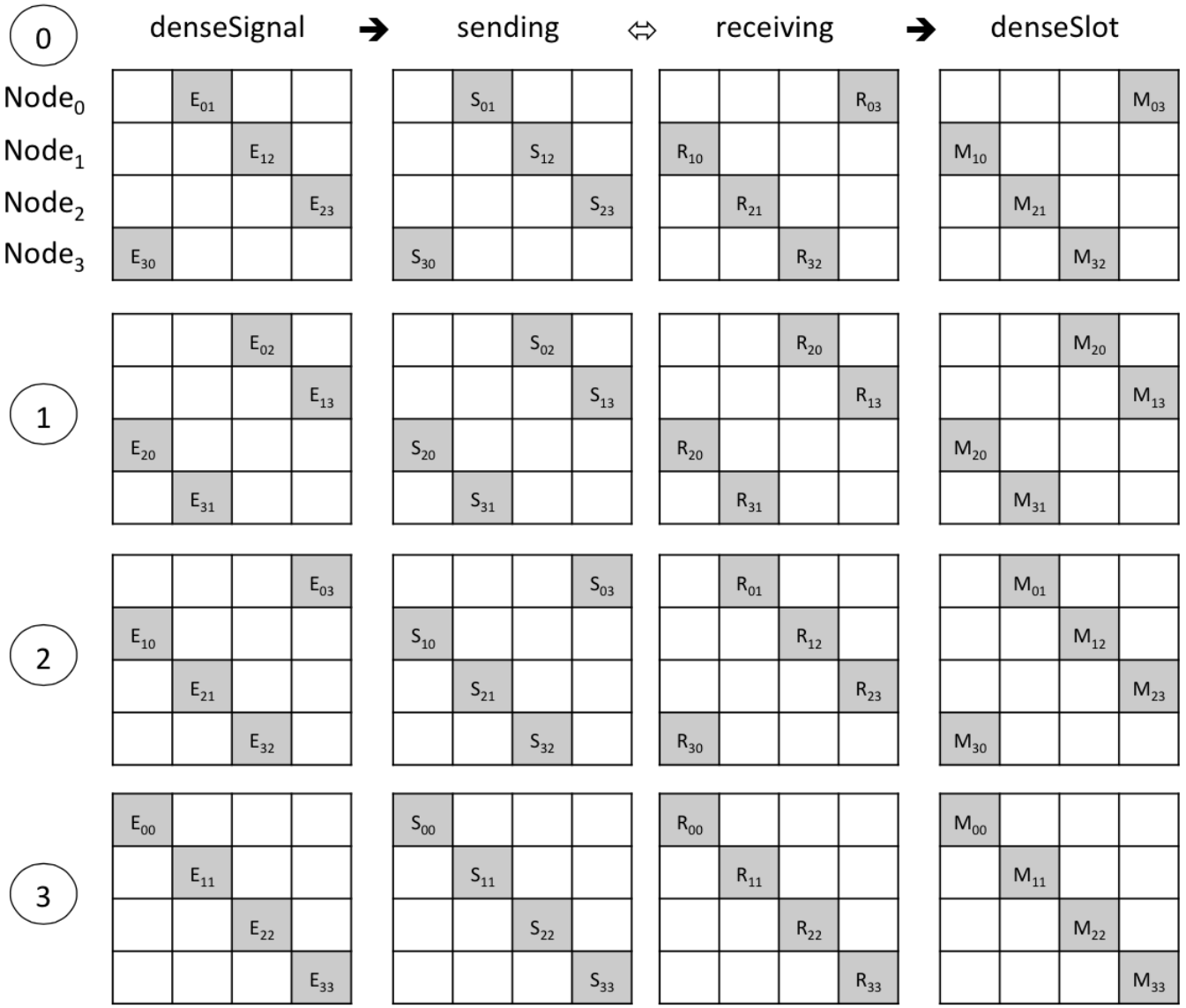
vtx = [1, 2, 3, 5]
offset = [0, 1, 2, 3, 4]

} CI

Nb: source vid
Idx:Number of edges of dest v given by idx[i+1]-idx[i]

- CSC representation requires O(|V|) memory in each nodes
  - May becomes bottleneck when number of partitions increases
  - Use two approaches to compress the data and use them adaptively

Computation and Communication Co-Scheduling



Dataset

Graph	**|V|**	**|E|**
EnWiki [1]	4.21M	101M
Twitter [16]	41.7M	1.47B
UK [6]	106M	3.74B
Weibo [13]	72.4M	6.43B
ClueWeb [2]	978M	42.6B
Kronecker [3, 18]	2^{Scale}	$2^{Scale+5}$

Gemini – Performance

- Test Environment
  - 8 dual Xeon E5-2670V3 4 (hypert.) vCPU cores
  - 128 GB memory
  - Infiniband EDR(100Gbps)

- 18.7X vs PowerLyra (PowerGraph)
- 80-300X vs GraphX (Spark)

Graph	PowerG.	GraphX	PowerL.	Gemini	Speedup (×times)
PR					
enwiki-2013	9.05	30.4	7.27	0.484	15.0
twitter-2010	40.3	216	26.9	3.76	7.15
uk-2007-05	64.9	416	58.9	1.48	39.8
weibo-2013	117	-	100	8.36	12.0
clueweb-12	-	-	-	36.9	n/a
CC					
enwiki-2013	4.61	16.5	5.02	0.237	19.5
twitter-2010	29.1	104	22.0	1.13	19.5
uk-2007-05	72.1	-	63.4	2.08	30.5
weibo-2013	56.5	-	58.6	2.62	21.6
clueweb-12	-	-	-	24.5	n/a
SSSP					
enwiki-2013	16.5	151	17.1	0.514	32.1
twitter-2010	12.5	108	10.8	1.15	9.39
uk-2007-05	117	-	143	3.45	33.9
weibo-2013	63.2	-	60.6	4.24	14.3
clueweb-12	-	-	-	44.3	n/a
GEOMEAN					18.7

Single node performance

System	Ligra	Galois	Gemini
Runtime (s)	44.5	19.3	12.7
Instructions	915G	482G	337G
Mem. Ref.	55.2G	23.4G	24.8G
IPC	0.389	0.405	0.435
LLC Miss	50.7%	49.7%	43.3%
CPU Util.	85.5%	96.8%	95.1%

Single node performance on PageRank - Twitter

Memory Usage

Graph	Raw	Gemini	PowerGraph
EnWiki	0.755	4.02	13.1
Twitter	10.9	25.1	138
UK	27.8	57.2	322
Weibo	47.9	73.3	561
ClueWeb	318	575	-

- Gemini uses less than 20% of PowerGraph
  - Less nodes are required to process a specific graph

Gemini

- Performance as the first design choice instead of fault-tolerance/scalability
  - 10X faster and 5X more memory efficient than current distributed graph systems
  - 100X faster than GraphX(1000X faster than naïve Spark)
  - Close to MPI performance with much friendly programming interfaces

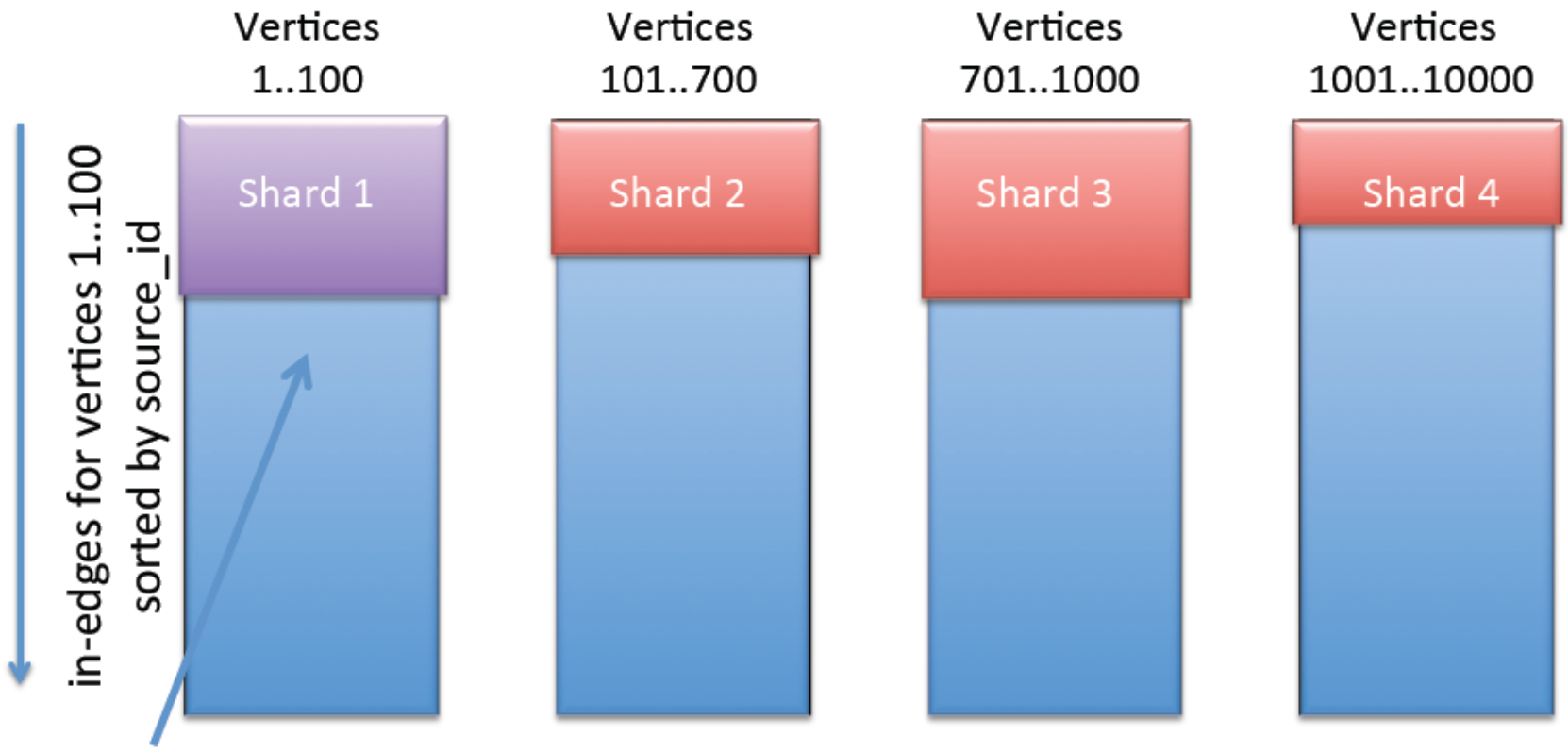
More on Single Machine Graph Systems

X-Stream and GridGraph

PSW: Loading Sub-graph

1. Load
2. Compute
3. Write

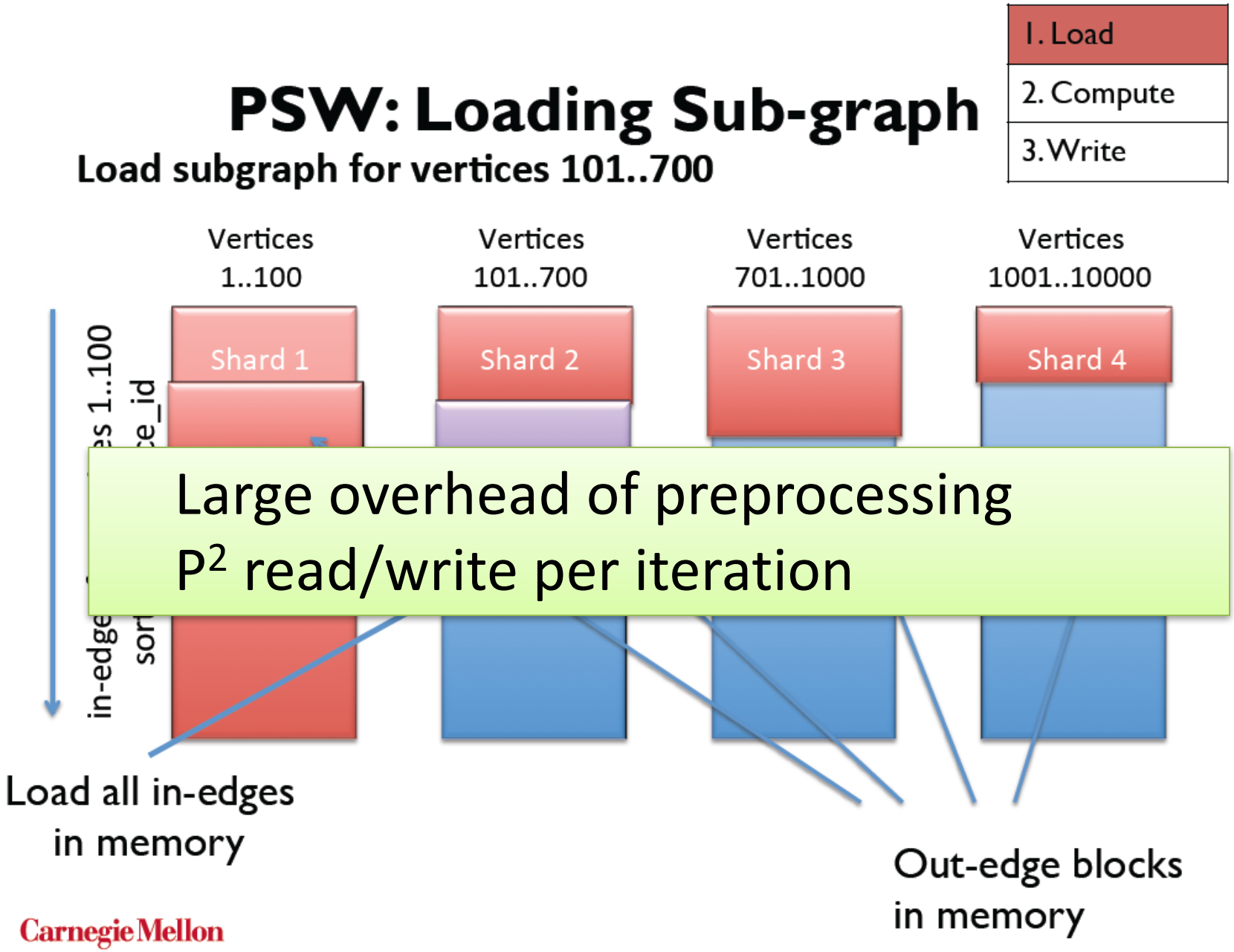
Load subgraph for vertices 1..100



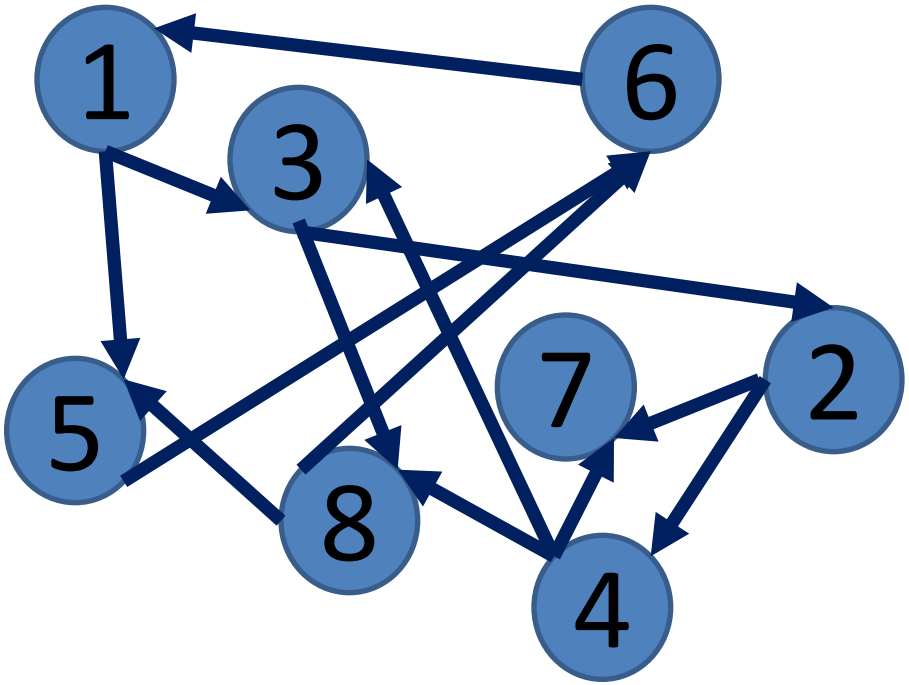
Load all in-edges in memory

What about out-edges?
Arranged in sequence in other shards

PSW: Loading Sub-graph



Standard Scatter Gather



BFS

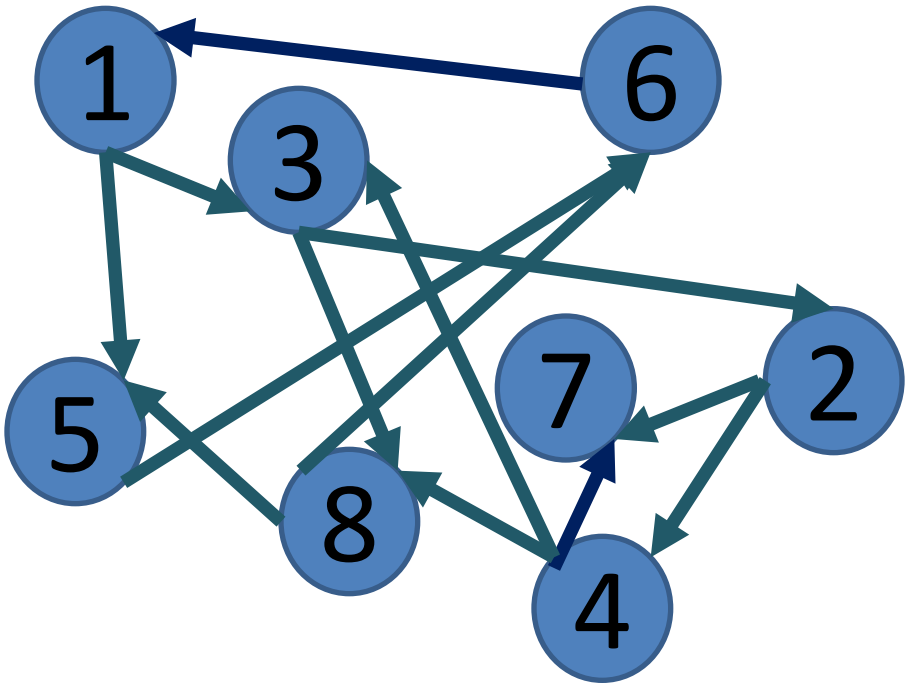
Vertex-Centric Scatter Gather

- Iterates over vertices

```
for each vertex v                    Scatter
   if v has update
      for each edge e from v
         scatter update along e
```

- Standard scatter gather is **vertex-centric**
- Does not work well with storage

Vertex-Centric
Scatter Gather



BFS

Lookup Index

V
1
2
3
4
5
6
7
8

SOURCE	DEST
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

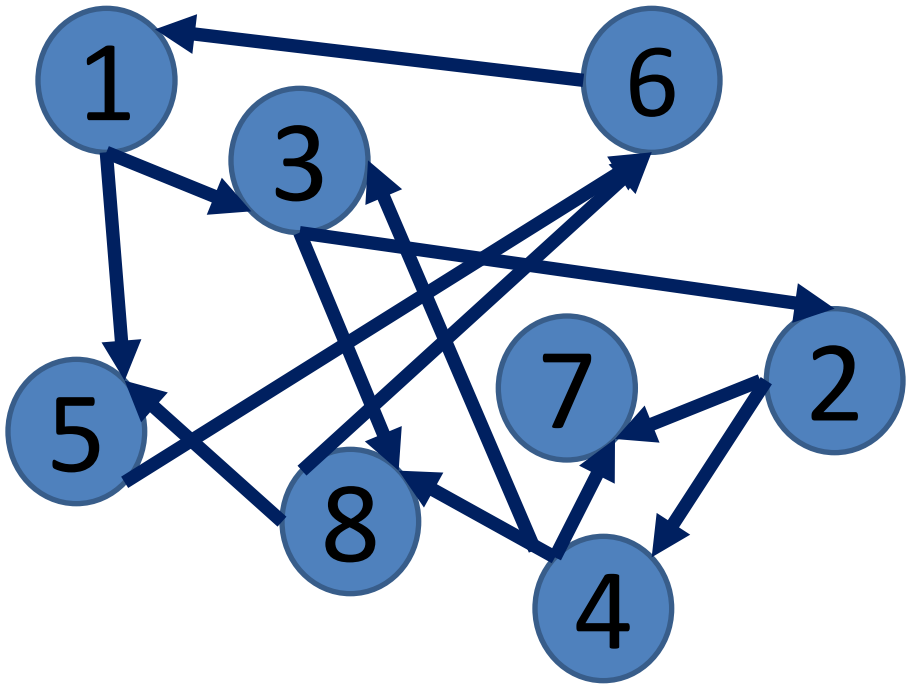
X-Stream : Edge-Centric model

```
for each vertex v                    Scatter
   if v has update
      for each edge e from v
         scatter update along e
```

```
for each edge e                      Scatter
   If e.src has update
         scatter update along   e
```

Vertex-Centric ⇨ **Edge-Centric**

Edge Centric Processing



BFS

V
1
2
3
4
5
6
7
8

SOURCE	DEST
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

SOURCE	DEST
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

=

SOURCE	DEST
1	3
8	6
5	6
2	4
3	2
4	7
4	3
3	8
4	8
2	7
6	1
8	5
1	5

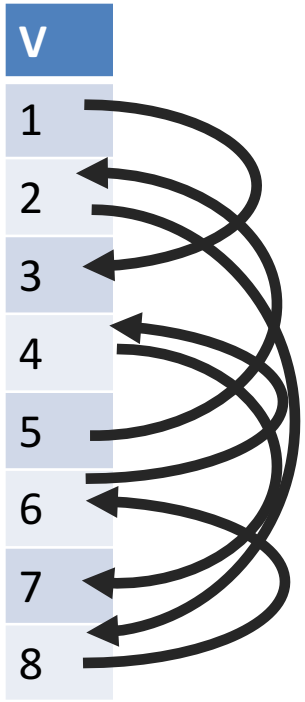
No index
No clustering
No sorting

Tradeoff

- Vertex-centric scatter-gather: Edge Data / Random Access bandwidth
- Edge-centric scatter-gather:Scatters x Edge Data/Sequential Access Bandwidth
- Sequential Access Bandwidth >> Random Access Bandwidth
- Few scatter gather iterations for real world graphs
  - Give an example that is not suitable for this system?

Streaming Partitions

- Problem: still have random access to vertex set



- Solution: partition the graph into streaming partitions

Streaming Partitions

- A streaming partition is
  - A subset of the vertices that fits in RAM
  - All edges whose source vertex is in that subset
  - No requirement on quality of the partition

Partitioning the Graph

Subset of vertices

V1
1
2
3
4

SOURCE	DEST
1	5
4	7
2	7
4	3
4	8
3	8
2	4
1	3
3	2

V2
5
6
7
8

SOURCE	DEST
5	6
8	6
8	5
6	1

Partition Edges and vertexs

- Shuffle edges with source vertex id(instead of sorting)
- Source vertex in memory

V1
1
2
3
4

SOURCE	DEST
1	5
4	7
2	7
4	3
4	8
3	8
2	4
1	3
3	2

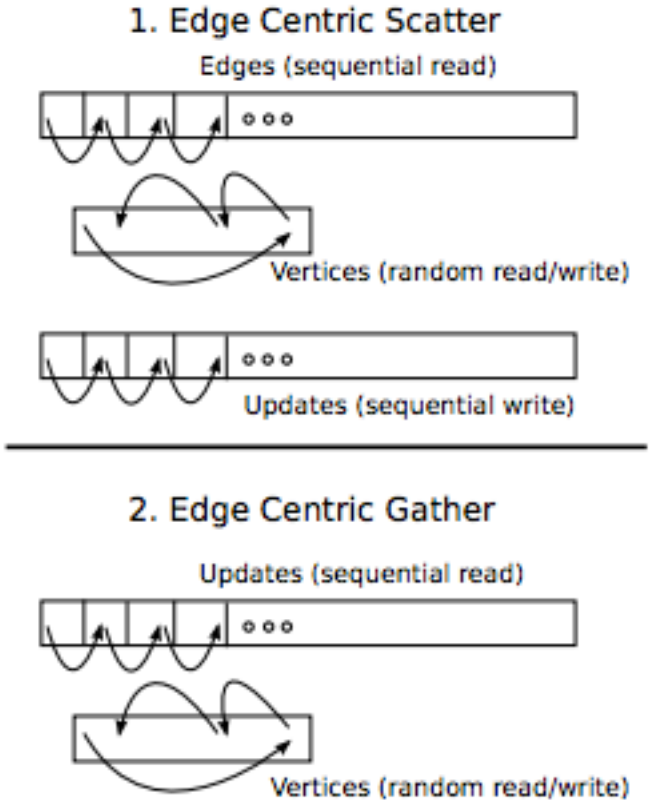
Edges in disks/SSDs

V2
5
6
7
8

SOURCE	DEST
5	6
8	6
8	5
6	1

X-Stream: Edge Centric Computing with Partitions

```
scatter phase:
  for each streaming_partition p
    read in vertex set of p
    for each edge e in edge list of p
      edge_scatter(e): append update to Uout

shuffle phase:
  for each update u in Uout
    let p = partition containing target of u
    append u to Uin(p)
  destroy Uout

gather phase:
  for each streaming_partition p
    read in vertex set of p
    for each update u in Uin(p)
      edge_gather(u)
    destroy Uin(p)
```



The problems of GraphChi and X-Stream

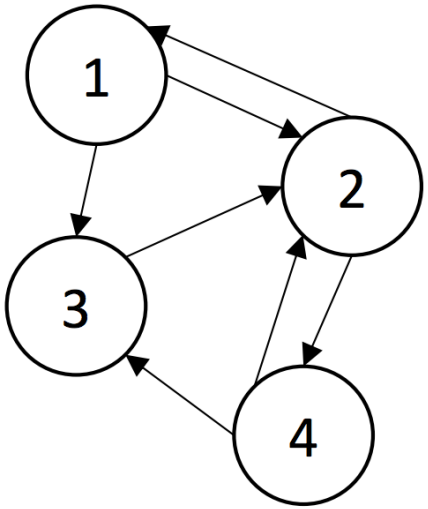
- GraphChi
  - Large overhead of preprocessing
  - P^2 read/write per iteration
- X-Stream
  - Separate gather/scatter phase, with shuffle between them, introduce many extra I/O operations
  - Poor performance for BFS and WCC-like algorithms
    - Only a small portion of edges are touched for some iterations, but they would scan all edges

GridGraph

- A 2-level partitioning graph data structure
  - Improve locality
  - Enable more effective streaming on both edges and vertices
- A Streaming-Apply model
  - Combines the gather-apply-scatter operations
- A programming abstraction
  - To enable the streaming-apply model

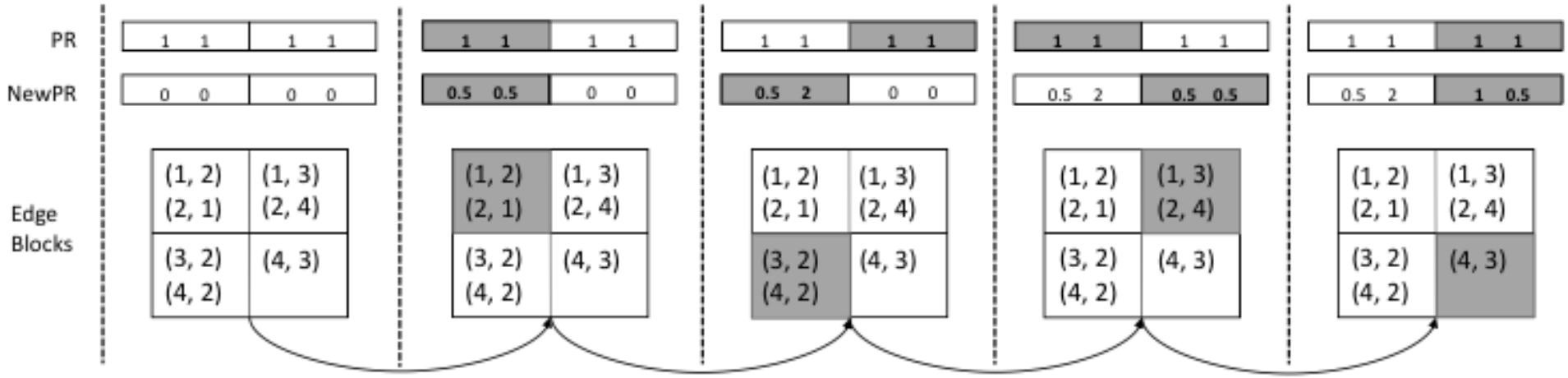
The Grid Representation

- 2-level hierarchical partitioning
  - 1st dimension
    - partition the vertices into chunks(1 2)(3,4)
    - partition the edges into shards by source vertex((1,2),(2,1),(1,3),(2,4))
  - 2nd dimension
    - partition the shards into blocks by destination vertex



(1, 2) (2, 1)	(1, 3) (2, 4)
(3, 2) (4, 2)	(4, 3)

The Streaming-Apply Model



Dual Sliding Windows:

- 1 read window + 1 write window
- Contents of active windows can be fit into RAM

Each iteration, just read edge list once, has opportunity to skip some blocks

Algorithm 1 Edge Block Streaming

```
for j ← 1, P do
    for i ← 1, P do
        if ChunkIsActive(i) then
            StreamEdgeBlock(i, j);
        end if
    end for
end for
```

Streaming-Apply Processing Model

- Vertex accesses are aggregated
  - Good locality
    - Only 2 partitions of vertices are accessed within each edge block
- On-the-fly updates onto vertices
  - Reduces I/O
  - Enables asynchronous implementation of algorithms (like WCC, etc.) which converges faster

Programing Abstraction

- Preliminaries
  - Atomic operations
    - CompareAndSwap
    - SetMinimum
    - Accumulate
  - BigVector
    - Memory mapped vertex arrays
    - Access using [] operator

Programming Abstraction

Algorithm 2 Edge Streaming Interface

```
function StreamEdges(F_e, F)
    Sum = 0
    for all block do                    ▷ with active vertices
        for all edge ∈ block do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
        end for
    end for
    return Sum
end function
```

←

StreamEdges(Fe, F):
Fe: process the edge, which is allowed to side-effect related vertex data
F: check whether the edge should be processed (by looking at the source vertex)

→

StreamVertices(Fv, F):
Fv: process the vertex
F: check if the vertex should be processed

Algorithm 3 Vertex Streaming Interface

```
function StreamVertices(F_v, F)
    Sum = 0
    for all vertex do
        if F(vertex) then
            Sum += F_v(edge)
        end if
    end for
    return Sum
end function
```

Programming Abstraction

BFS

Algorithm 4 BFS

```
function IsActive(v)
    return ActiveIn[v]
end function
function Visit(e)
    if CAS(&Parent[e.dest], -1, e.source) then
        ActiveOut[e.dest] = 1
        return 1
    end if
    return 0
end function
ActiveVertices = 1
Parent = {-1, ..., -1}
ActiveIn = {0, ..., 0}
Parent[start] = start
ActiveIn[start] = 1
while ActiveVertices > 0 do
    ActiveOut = {0, ..., 0}
    ActiveVertices = StreamEdges(Visit, IsActive)
    Swap(ActiveIn, ActiveOut)
end while
```

PageRank Implementation

Algorithm 7 PageRank

```
function Contribute(e)
    Accum(&NewPR[e.dest], PR[e.source] / Degree[e.source])
end function
function Compute(v)
    NewPR[i] = (1-Gamma) + Gamma*NewPR[v]
    return abs(NewPR[i] - PR[i])
end function
Gamma = 0.85
PR = {1, ..., 1}
Converged = 0
while Converged do
    NewPR = {0, ..., 0}
    StreamEdges(Contribute, FTrue)
    Diff = StreamVertices(Compute, FTrue)
    Swap(PR, NewPR)
    Converged = Diff / V ≤ Threshold
end while
```

Evaluation

- Datasets
  - Real world datasets
    - Social graph: LiveJournal, Twitter
    - Web graph: UK, Yahoo

Dataset	|V|	|E|	Data size
LiveJournal	4.85 million	69.0 million	527MB
Twitter	61.6 million	1.47 billion	11GB
UK	106 million	3.74 billion	28GB
Yahoo	1.41 billion	6.64 billion	50GB

- Benchmarks
  - BFS, WCC, SpMV, PageRank

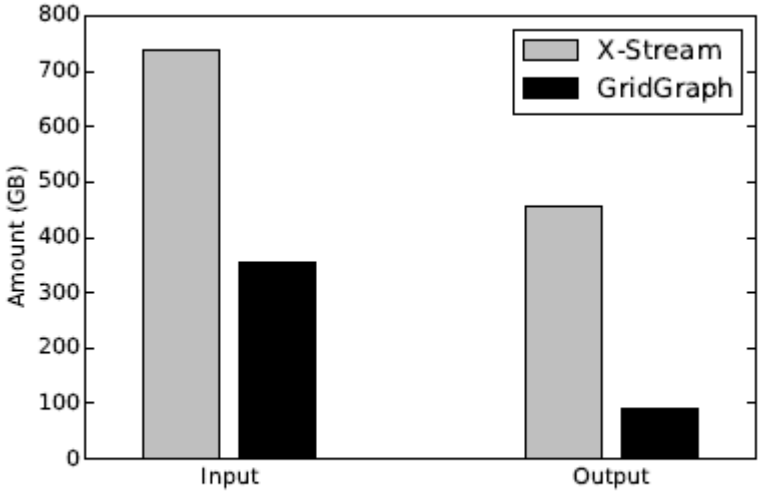
Performance vs. GraphChi, X-Stream

- Test environment
  - 1 AWS i2.xlarge instance
    - 4 (hypert.) vCPU cores
    - 30.5 GB memory
      - Memory limited to 8 GB
    - 800GB SSD

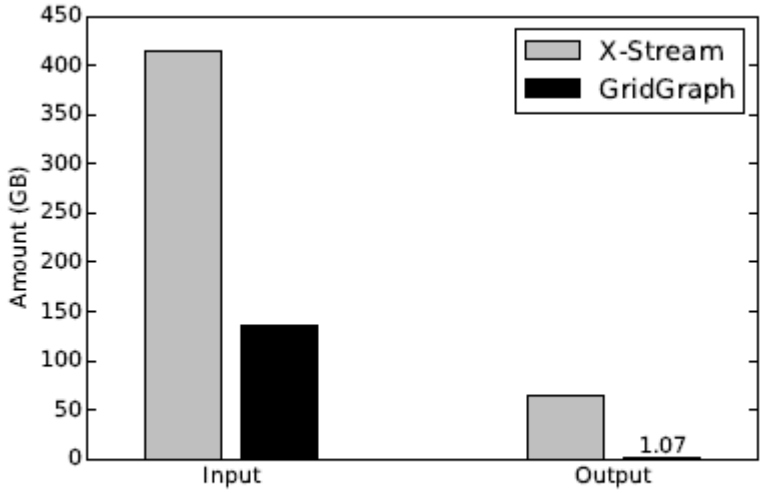
Runtime

	BFS	WCC	SpMV	PageR.
LiveJournal				
GraphChi	22.05	17.28	10.12	52.08
X-Stream	6.54	14.65	6.63	18.22
GridGraph	2.11	2.53	1.96	10.54
Twitter				
GraphChi	411.3	439.6	254.0	1225
X-Stream	435.9	1199	143.9	1779
GridGraph	51.34	190.3	43.78	461.4
UK				
GraphChi	3776	2527	407.2	3307
X-Stream	8081	12057	383.7	4374
GridGraph	979.0	1264	106.3	1285
Yahoo				
GraphChi	-	-	1540	13416
X-Stream	-	-	1076	9957
GridGraph	11935	3694	379.0	3923

I/O Amount vs. X-Stream



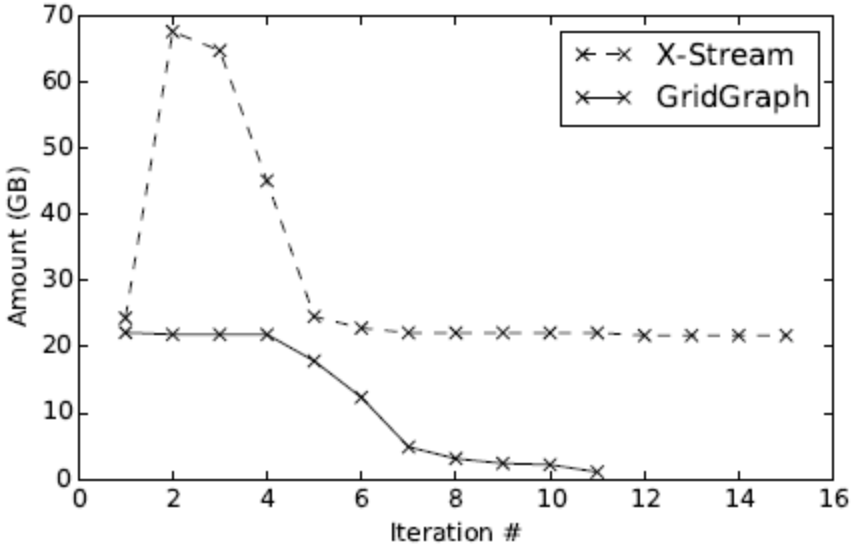
(a) PageRank on Yahoo graph with 5 iterations



(b) WCC on Twitter graph

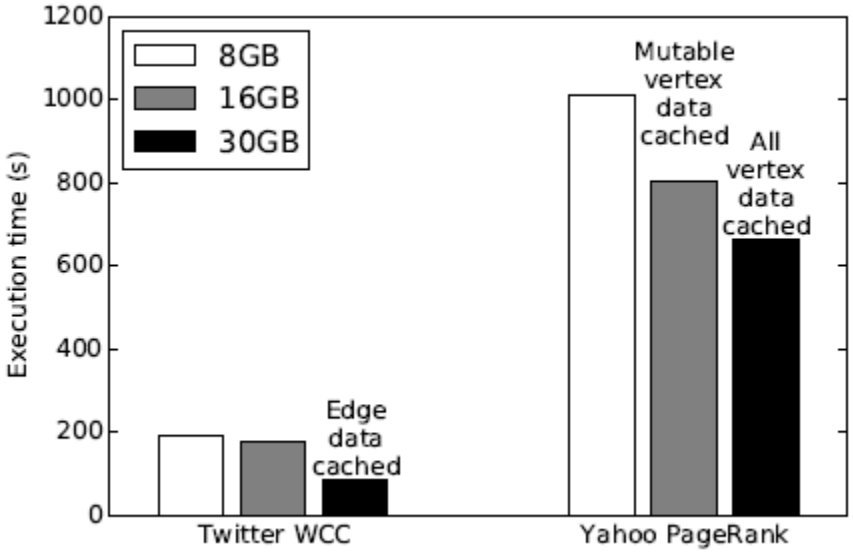
- Effects of Streaming-Apply and selective scheduling
- Less I/O (especially write) not only good for performance, but also good for life time of SSDs

I/O Amount vs. X-Stream



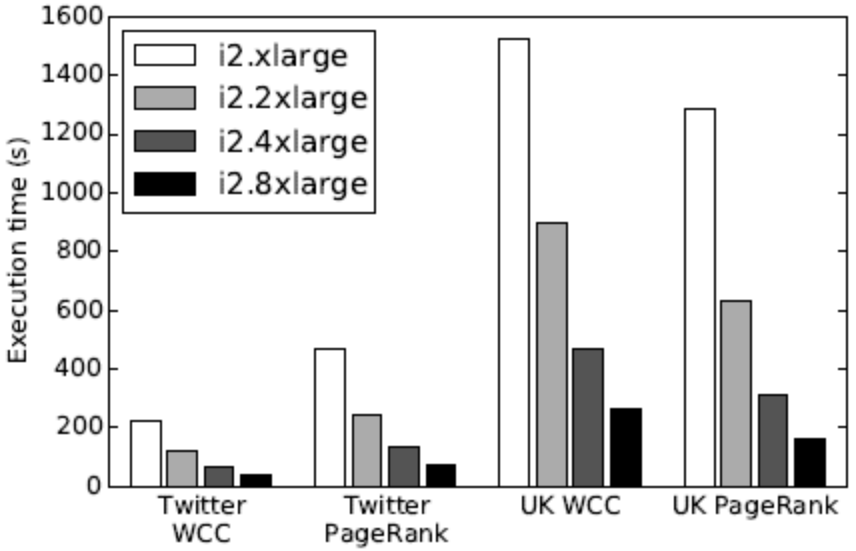
Faster convergence due to “asynchronous” label propagation
WCC on Twitter Graph

Scalability



← Scalability with available memory size

Scalability with I/O throughput →



Performance vs. PowerGraph, GraphX

- Test environment
  - PowerGraph, GraphX
    - 16 AWS m2.4xlarge instances
  - GridGraph
    - 1 AWS i2.4xlarge instance(4 SSDs)

System	Twitter WCC	Twitter PR	UK WCC	UK PR	Cost per hr.
PowerGraph	244	249	714	833	15.68
GraphX	251	419	647	462	15.68
GridGrpah	64	132	471	314	3.41

Conclusion

- Graph computing is an interesting new paradigm in big data era
- Unstructured data gives challenges to system and algorithm designers
  - Random access
  - Load imbalance in a dynamic way
- Different system approaches
  - Data layout and partition
  - Leverage the associativity
  - Batching