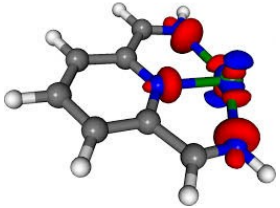
GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning

Xiaowei ZHU

Tsinghua University

Widely-Used Graph Processing



Existing Solutions

- Shared memory
  - Single-node & in-memory
  - Ligra, Galois, Polymer
- Distributed
  - Multi-node & in-memory
  - GraphLab, GraphX, PowerLyra
- Out-of-core
  - Single-node & disk-based
  - GraphChi, X-Stream, TurboGraph

Existing Solutions

- Shared memory
  - Single-node & in-memory
  - Ligra, Galois, Polymer
- Distributed
  - Multi-node & in-memory
  - GraphLab, GraphX, PowerLyra
- Out-of-core
  - Single-node & disk-based
  - GraphChi, X-Stream, TurboGraph

Large-scale
Limited capability to big graphs

Irregular structure
Imbalance of computation and communication

Inevitable random access
Expensive disk random access

Existing Solutions

- Shared memory
  - Single-node & in-memory
  - Ligra, Galois, Polymer
- Distributed
  - Multi-node & in-memory
  - GraphLab, GraphX, PowerLyra
- Out-of-core
  - Single-node & disk-based
  - GraphChi, X-Stream, TurboGraph

Most cost effective!

Large-scale
Limited capability to big graphs

Irregular structure
Balance of computation and communication

Inevitable random access
Expensive disk random access

Methodology

- How to handle graphs that is much larger than memory capacity?
  - Partition!

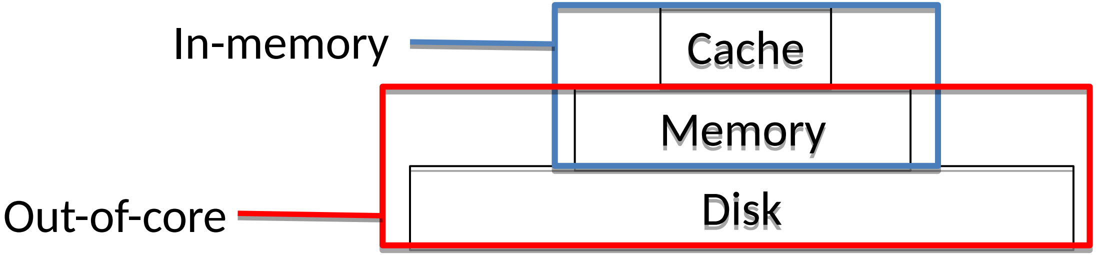
System	Unit
GraphChi	Shards
TurboGraph	Page
X-Stream	Streaming Partitions
PathGraph	Tree-Based Partitions
FlashGraph	Page
GridGraph	Chunks & Blocks

State-of-the-Art Methodology

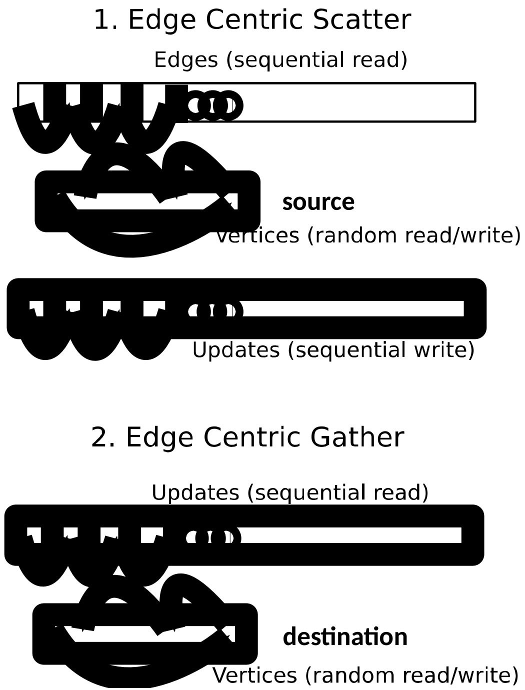
- X-Stream
  - Access edges **<u>sequentially</u>** from **<u>disks</u>**
  - Access vertices **<u>randomly</u>** inside **<u>memory</u>**
    - Guarantee locality of vertex accesses by partitioning

State-of-the-Art Methodology

- X-Stream
  - Access edges **sequentially** from **slow** memory
  - Access vertices **randomly** inside **fast** memory
    - Guarantee locality of vertex accesses by partitioning



Edge-Centric Scatter-Gather

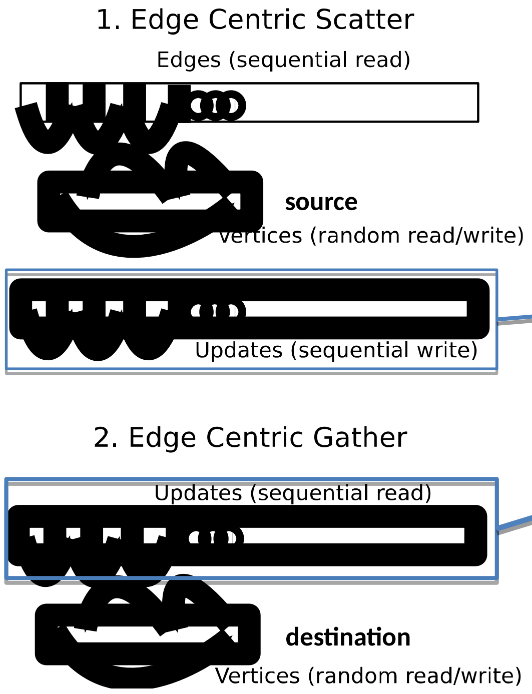


Scatter:
for each streaming partition
    load **source** vertex chunk of edges into fast memory
    stream **edges**
        append to several **updates**

Gather:
for each streaming partition
    load **destination** vertex chunk of updates into fast memory
    stream **updates**
        apply to vertices

X-Stream: Edge-centric Graph Processing using Streaming Partitions, A. Roy et al., SOSP 2013

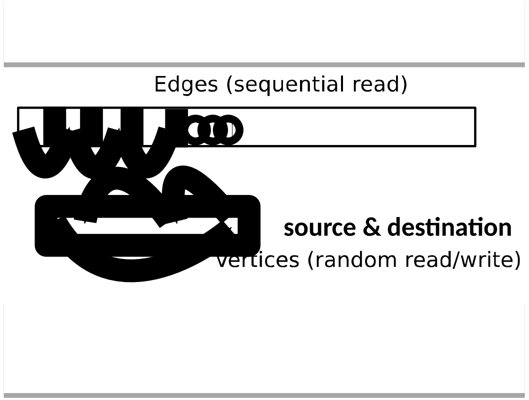
Motivation



Question: Is it possible to apply on-the-fly updates?
(Thus bypass the writes and reads of updates.)

Can be as large as O(E)!

Basic Idea



Answer: Guarantee the locality of both **source** and **destination** vertices when streaming **edges**!

Streaming-Apply:

for each streaming edge block

  load **source** and **destination** vertex chunk of edges into memory

  stream **edges**

    read from **source** vertices

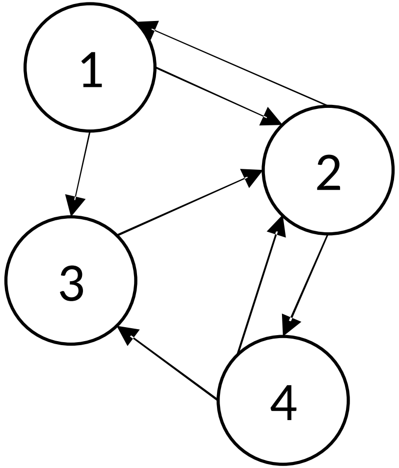
    write to **destination** vertices

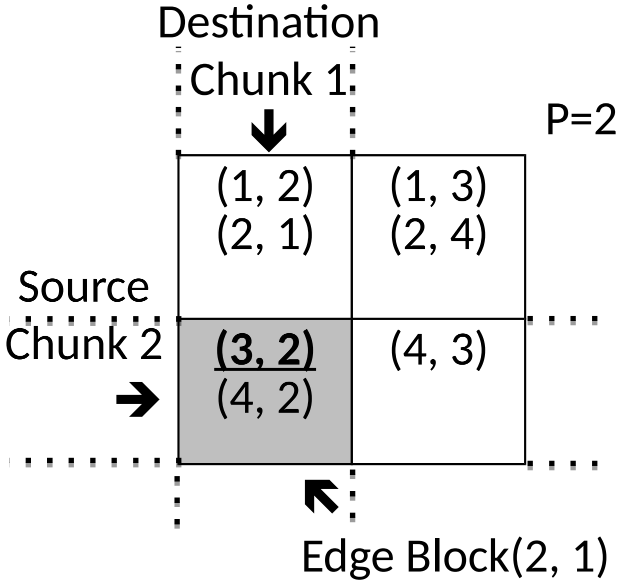
Solution

- Grid representation
  - Dual sliding windows
  - Selective scheduling
- 2-level hierarchical partitioning

Grid Representation

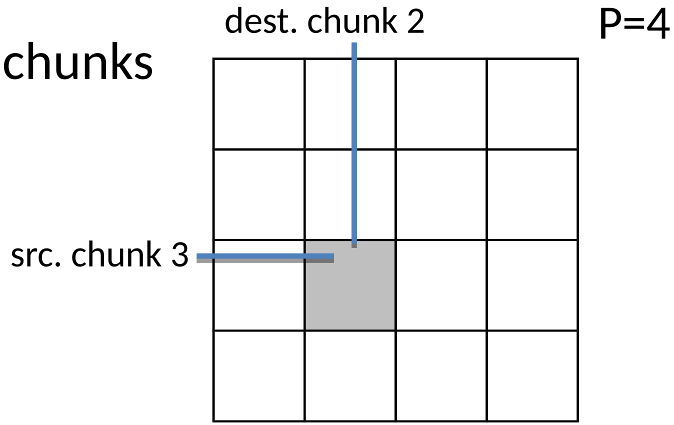
- Vertices partitioned into P equalized chunks
- Edges partitioned into P × P blocks
  - Row ⇔ source
  - Column ⇔ destination





Streaming-Apply Processing Model

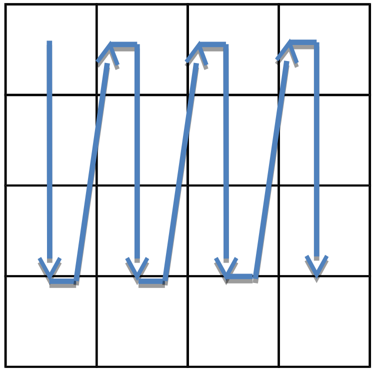
- Stream edges block by block
  - Each block corresponding to two vertex chunks
    - Source chunk + destination chunk
    - Fit into memory
- Difference with scatter-gather
  - 2 phases → 1 phase
  - Updates are applied on-the-fly



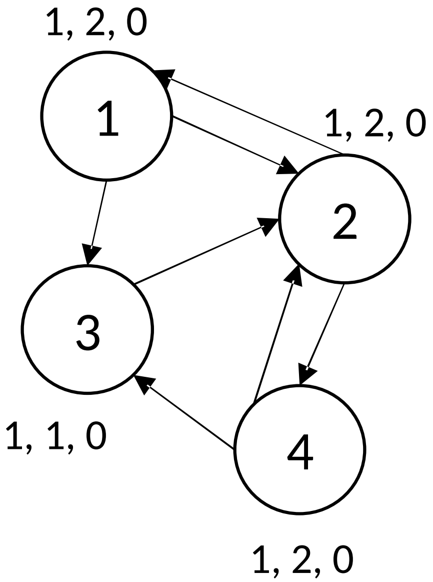
Dual Sliding Windows

- Access edge blocks in column-oriented order
  - From left to right
    - Destination window slides as column moves
  - From top to bottom
    - Source window slides as row moves
  - Optimize write amount
    - 1 pass over the destination vertices

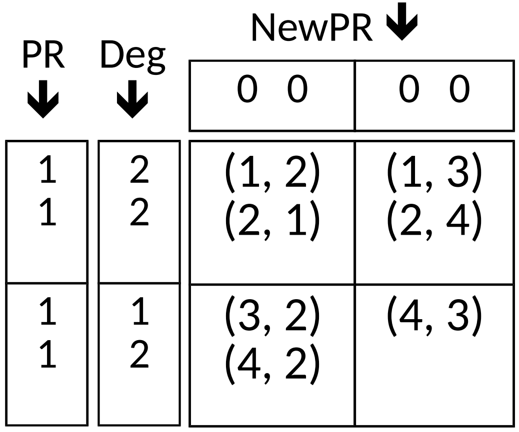
P=4



Dual Sliding Windows



P=2

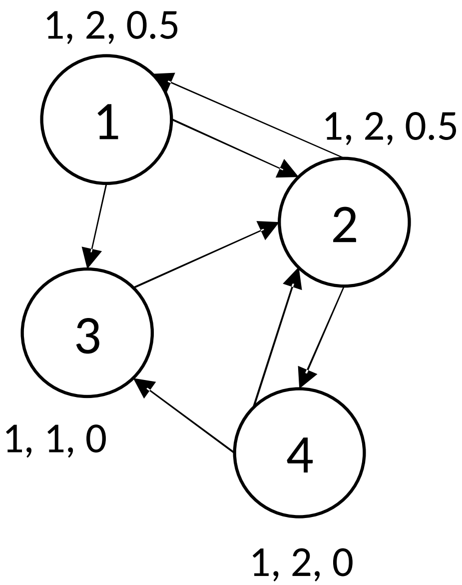


Initialize

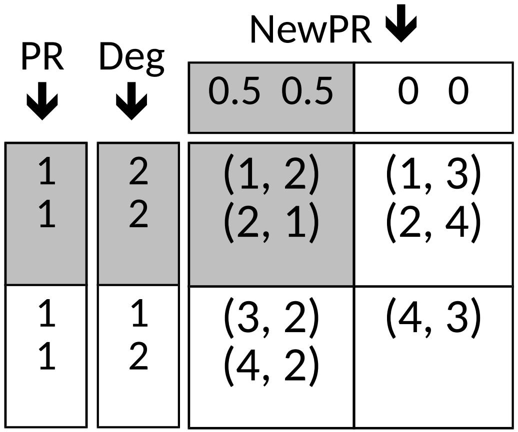
Object	I/O Amt.
Edges	0
Src. vertex	0
Dest. vertex	0

$$PageRank_i = (1-d) + d * \sum_{j \in N_{in}(i)} \frac{PageRank_j}{OutDegree_j}$$

Dual Sliding Windows



P=2

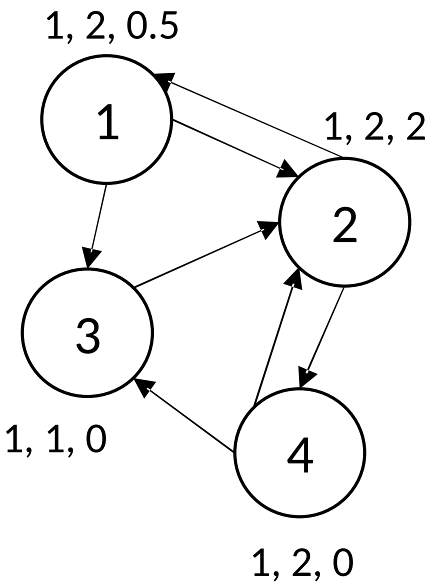


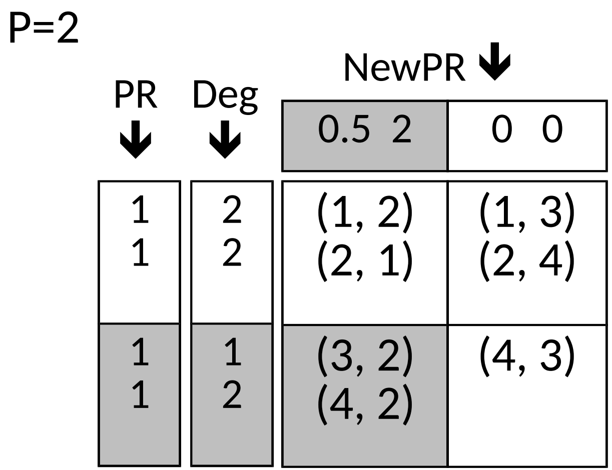
Stream Block (1, 1)

Object	I/O Amt.
Edges	0 → 2
Src. vertex	0 → 2
Dest. vertex	0 → 2

Cache source vertex chunk 1 in **memory** (**miss**);
Cache destination vertex chunk 1 in **memory** (**miss**);
Read edges (1, 2), (2, 1) from **disk**;

Dual Sliding Windows



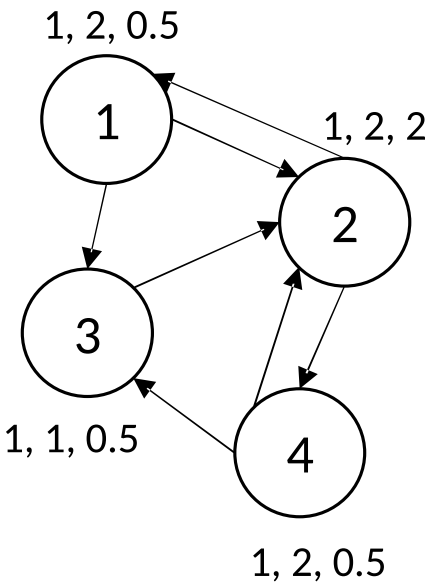


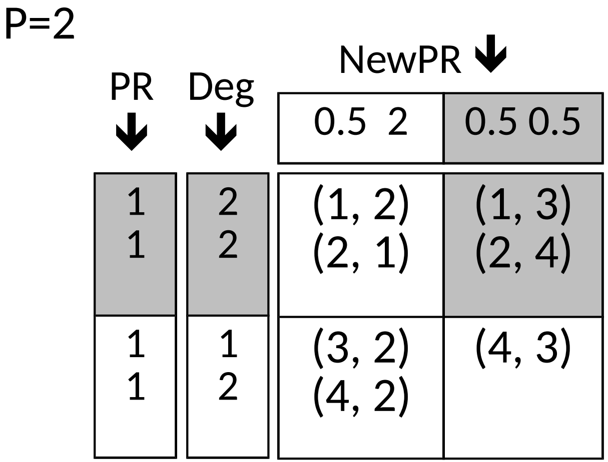
Stream Block (2, 1)

Object	I/O Amt.
Edges	2 → 4
Src. vertex	2 → 4
Dest. vertex	2

Cache source vertex chunk 2 in **memory** (**miss**);
Cache destination vertex chunk 1 in **memory** (**hit**);
Read edges (3, 2), (4, 2) from **disk**;

Dual Sliding Windows



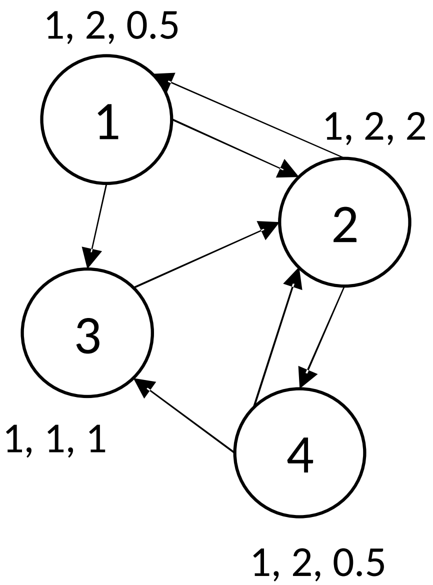


Stream Block (1, 2)

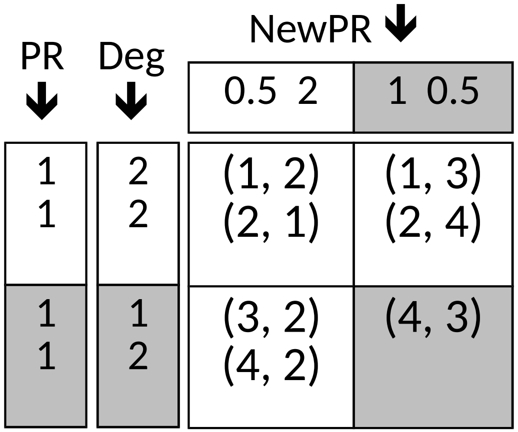
Object	I/O Amt.
Edges	4 → 6
Src. vertex	4 → 6
Dest. vertex	2 → 6

Cache source vertex chunk 1 in **memory** (**miss**);
Cache destination vertex chunk 2 in **memory** (**miss**);
Write back destination vertex chunk 1 to **disk**;
Read edges (1, 3), (2, 4) from **disk**;

Dual Sliding Windows



P=2

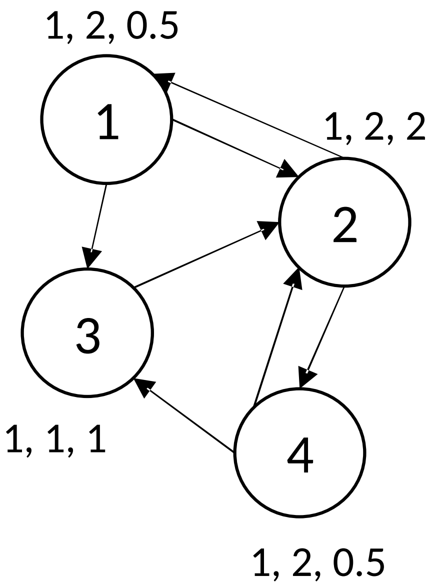


Stream Block (2, 2)

Object	I/O Amt.
Edges	6 → 7
Src. vertex	6 → 8
Dest. vertex	6

Cache source vertex chunk 2 in **memory** (**miss**);
Cache destination vertex chunk 2 in **memory** (**hit**);
Read edges (4, 3) from **disk**;

Dual Sliding Windows



P=2

PR	Deg	NewPR	
		0.5 2	1 0.5
1 1	2 2	(1, 2) (2, 1)	(1, 3) (2, 4)
1 1	1 2	(3, 2) (4, 2)	(4, 3)

Object	I/O Amt.
Edges	7
Src. vertex	8
Dest. vertex	6 → 8

Iteration 1 finishes

Write back destination vertex chunk 2 to **<u>disk</u>**;

I/O Access Amount

- For 1 iteration

$$\underline{E} + (\underline{2} + \underline{P}) \times V$$

1 pass over the edges (read)

P pass over the source vertices (read)

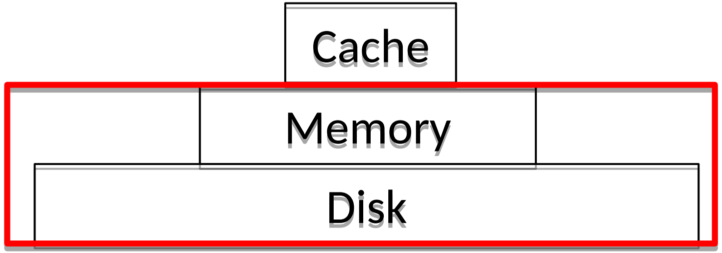
1 pass over the destination vertices (read+write)

Implication: P should be the minimum value that enables needed vertex data to be fit into memory.

I/O Access Amount

- For 1 iteration

$$E + (2 + P) \times V$$



1 pass over the edges (read)

P pass over the source vertices (read)

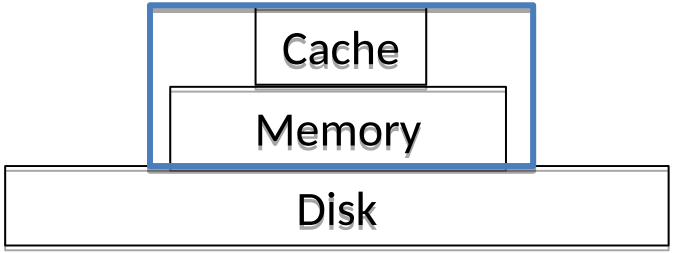
1 pass over the destination vertices (read+write)

Implication: P should be the minimum value that enables needed vertex data to be fit into memory.

Memory Access Amount

- For 1 iteration

$$E + (2 + P) \times V$$



1 pass over the edges (read)

P pass over the source vertices (read)

1 pass over the destination vertices (read+write)

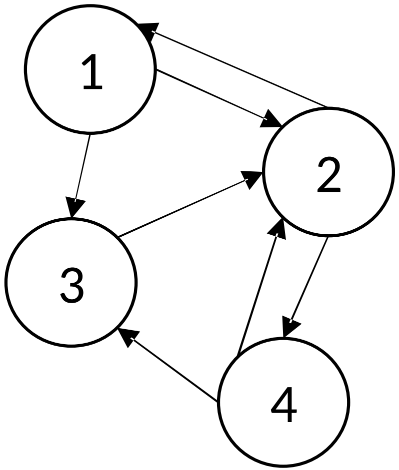
Implication: P should be the minimum value that enables needed vertex data to be fit into memory.

Selective Scheduling

- Skip blocks with no active edges
  - Very simple but important optimization
  - Effective for lots of algorithms
    - BFS, WCC, ...

Selective Scheduling

BFS from 1 with P = 2



Active	True	False	False	False
Parent	1	-1	-1	-1

Before

Iteration 1

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

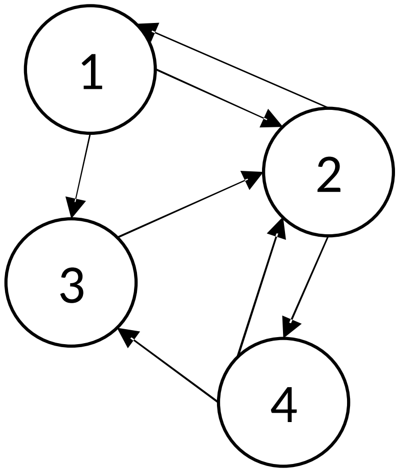
Access 4 edges

Active	False	True	True	False
Parent	1	1	1	-1

After

Selective Scheduling

BFS from 1 with P = 2



Before

Active	False	True	True	False
Parent	1	1	1	-1

Iteration 2

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

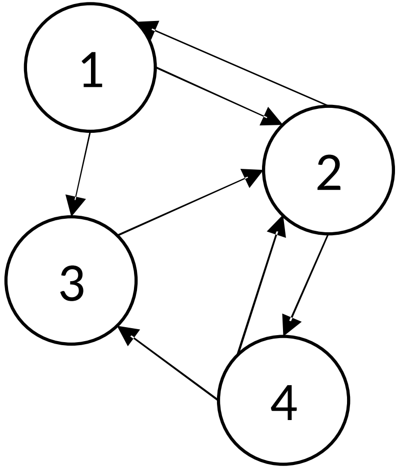
Access 7 edges

After

Active	False	False	False	True
Parent	1	1	1	2

Selective Scheduling

BFS from 1 with P = 2



Active	False	False	False	True
Parent	1	1	1	2

Before

Iteration 3

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

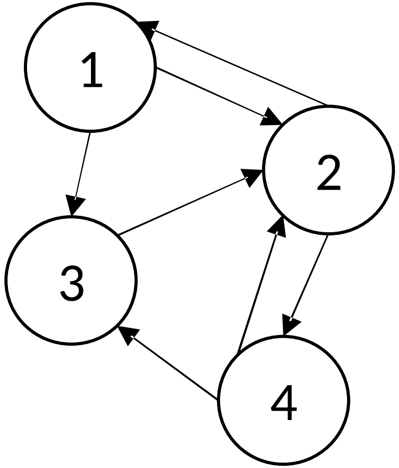
Access 3 edges

Active	False	False	False	False
Parent	1	1	1	2

After

Selective Scheduling

BFS from 1 with P = 2



Parent	1	1	1	2

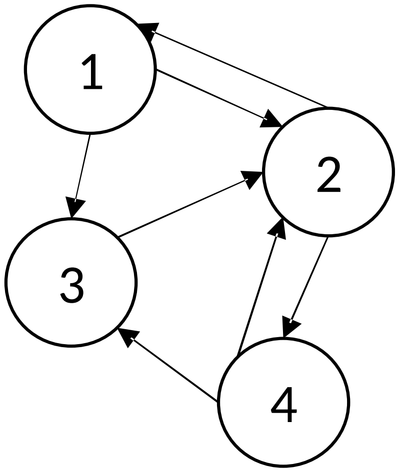
BFS finishes

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

4+7+3=14
Access 14 edges in all

Impact of P on Selective Scheduling

BFS from 1



P	1	2	4
Edge accesses	21(=7+7+7)	14=(4+7+3)	7=(2+3+2)

Effect becomes better with more fine-grained partitioning.

Implication: A larger value of P is preferred.

Dilemma on Selection of P

Coarse-grained

Fewer accesses on vertices

Poorer locality
Less selective scheduling

Fine-grained

Better locality
More selective scheduling

More accesses on vertices

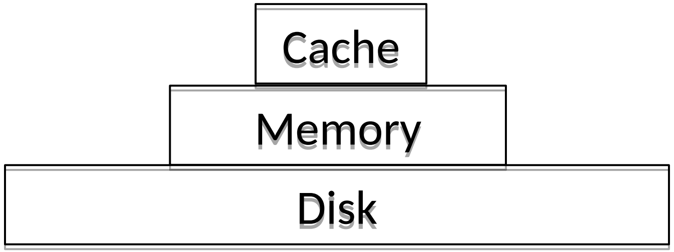
P

small

large

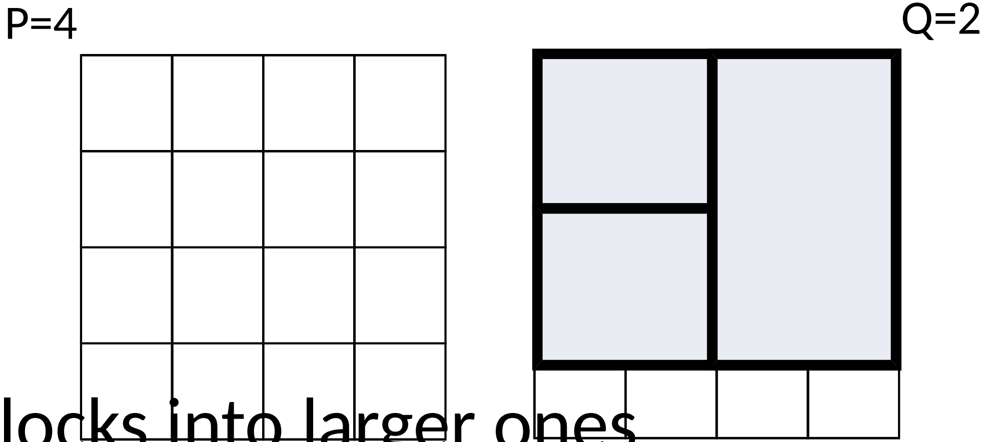
Dilemma from Memory Hierarchy

- Different selections of P
  - Disk – Memory hierarchy
    - Fit hot vertex data into memory
  - Memory – Cache hierarchy
    - Fit hot vertex data into cache
  - Disk – Memory – Cache
    - ?



2-Level Hierarchical Partitioning

- Apply a $Q \times Q$ partitioning over the $P \times P$ grid
    - $Q \geq V / M$
    - $P \geq V / C$
    - $C \ll M$
    - $P \gg Q$
  - Group the small blocks into larger ones



P = number of partitions, M = memory capacity, C = LLC capacity, V = size of vertices

Programming Interface

- StreamVertices(F_v, F)
- StreamEdges(F_e, F)

Algorithm 3 PageRank

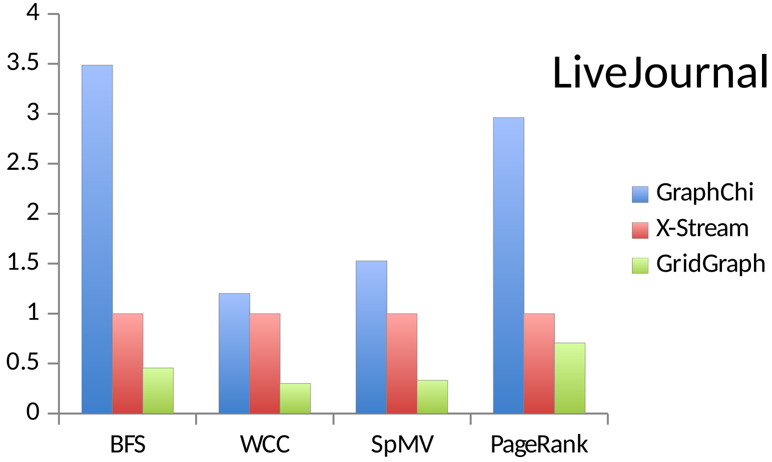
function CONTRIBUTE(e)
  Accum(&$NewPR[e.dest]$, $\frac{PR[e.source]}{Deg[e.source]}$)
end function
function COMPUTE(v)
  $NewPR[v] = 1 - d + d \times NewPR[v]$
  return $|NewPR[v] - PR[v]|$
end function
$d = 0.85$
$PR = \{1, ..., 1\}$
$Converged = 0$
while $\neg Converged$ **do**
  $NewPR = \{0, ..., 0\}$
  StreamEdges(Contribute)
  $Diff$ = StreamVertices(Compute)
  Swap(PR, $NewPR$)
  $Converged = \frac{Diff}{V} \leq Threshold$
end while

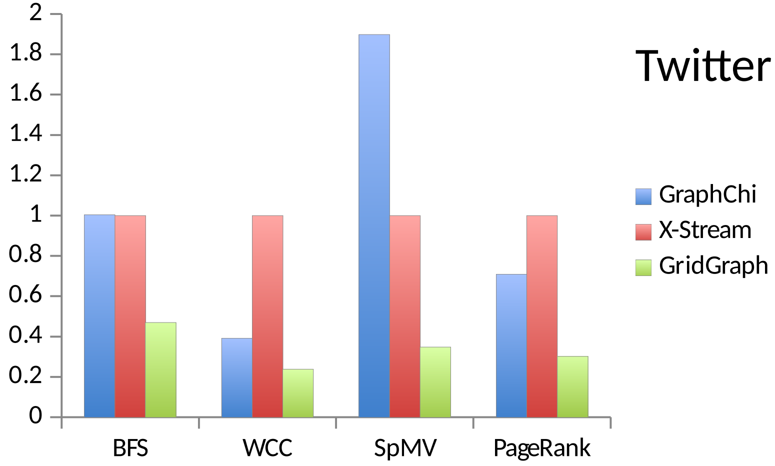
Evaluation

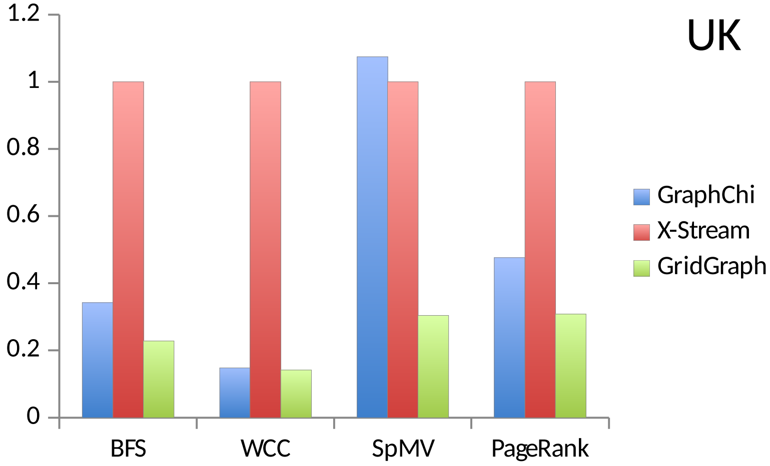
- Test environment
  - AWS EC2 i2.xlarge
    - 4 hyperthread cores
    - 30.5GB memory
    - 1 × 800GB SSD
  - AWS EC2 d2.xlarge
    - 4 hyperthread cores
    - 30.5GB memory
    - 3 × 2TB HDD

- Applications
  - BFS, WCC, SpMV, PageRank

Dataset	V	E	Data size	P
LiveJournal	4.85M	69.0M	527 MB	4
Twitter	61.6M	1.47B	11 GB	32
UK	106M	3.74B	28 GB	64
Yahoo	1.41B	6.64B	50 GB	512







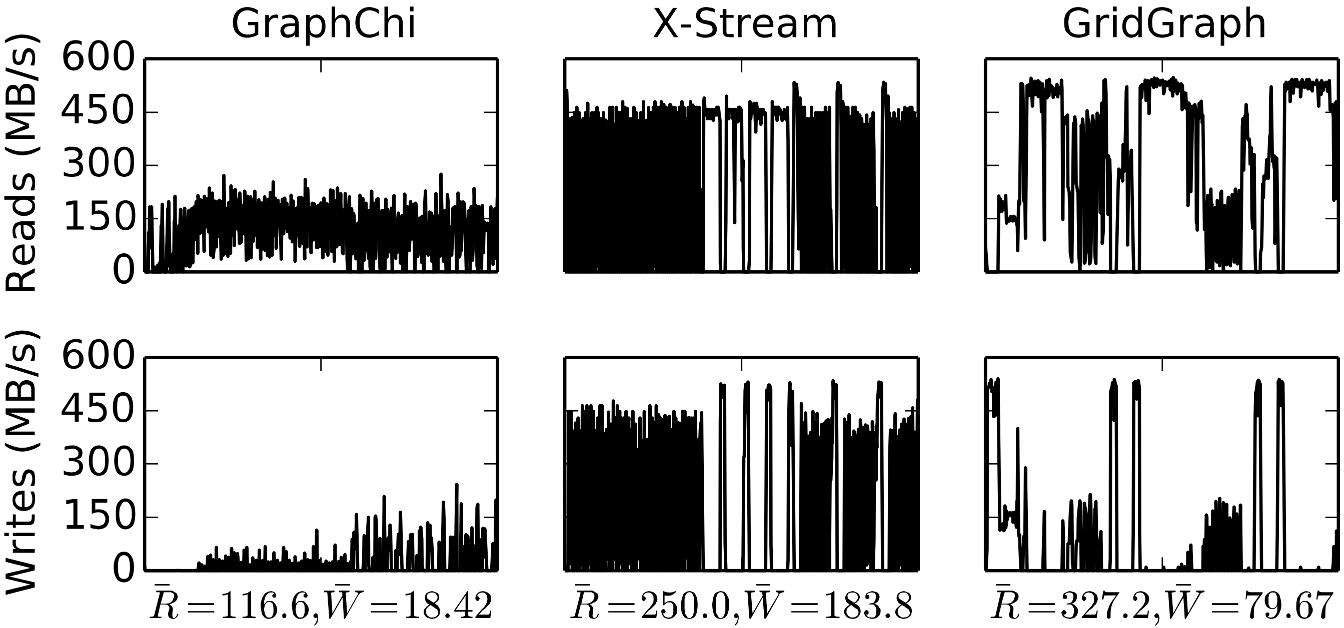
Yahoo

Runtime(S)	BFS	WCC	SpMV	PageRank
GraphChi	-	114162	2676	13076
X-Stream	-	-	1076	9957
GridGraph	16815	3602	263.1	4719

“-” indicates failing to finish in 48 hours

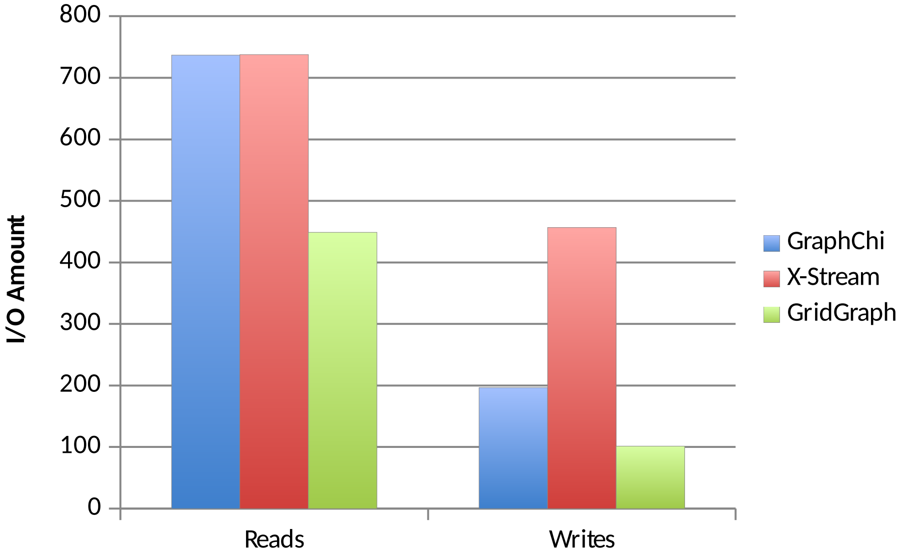
i2.xlarge, memory limited to 8GB

Disk Bandwidth Usage



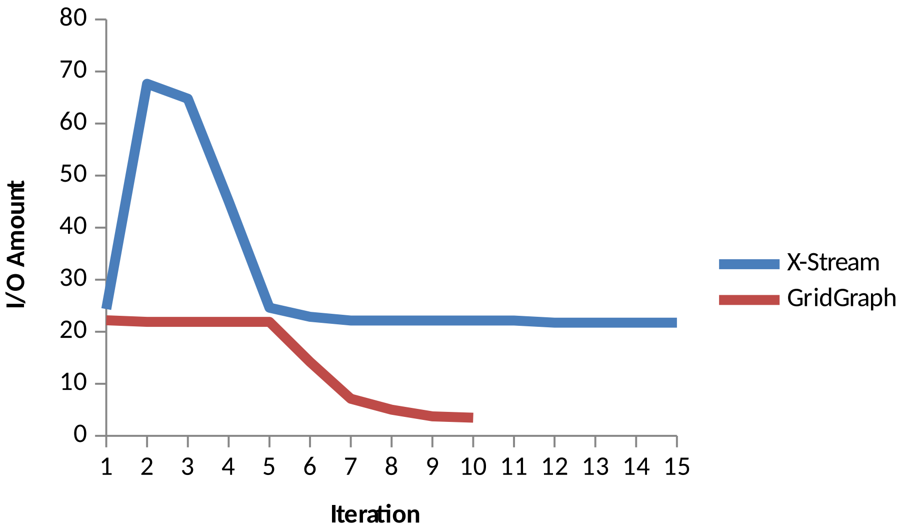
I/O throughput of a 10-minute interval running PageRank on Yahoo graph

Effect of Dual Sliding Windows



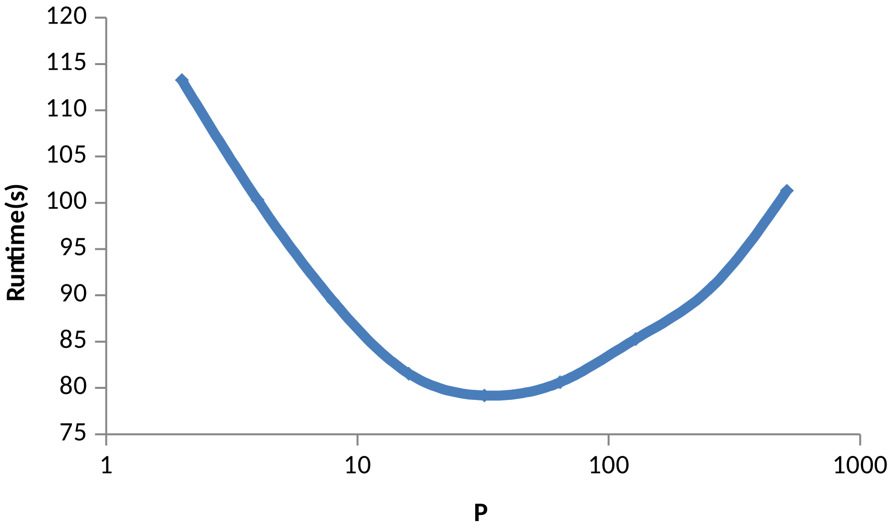
PageRank on Yahoo

Effect of Selective Scheduling



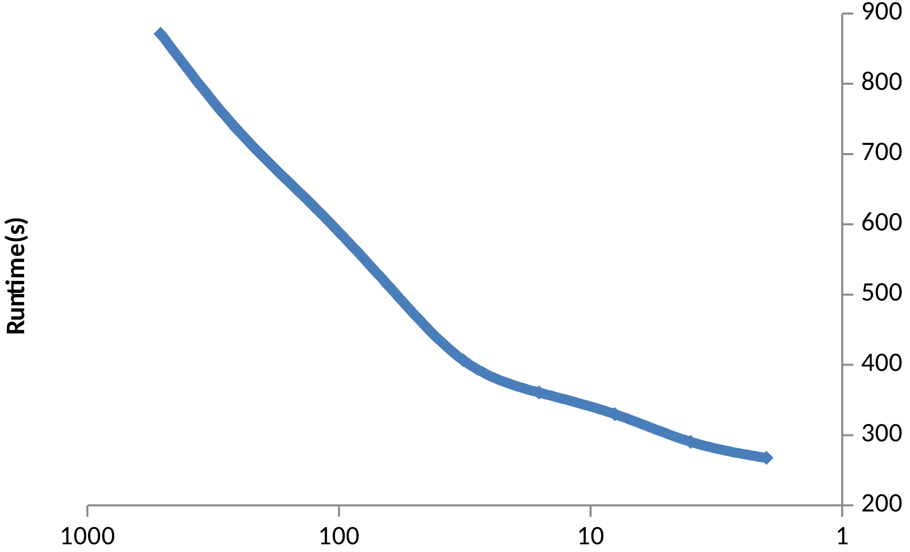
WCC on Twitter

Impact of P on In-Memory Performance



PageRank on Twitter
Edges cached in 30.5GB memory
Optimal around P=32 where needed vertex data can be fit into L3 cache.

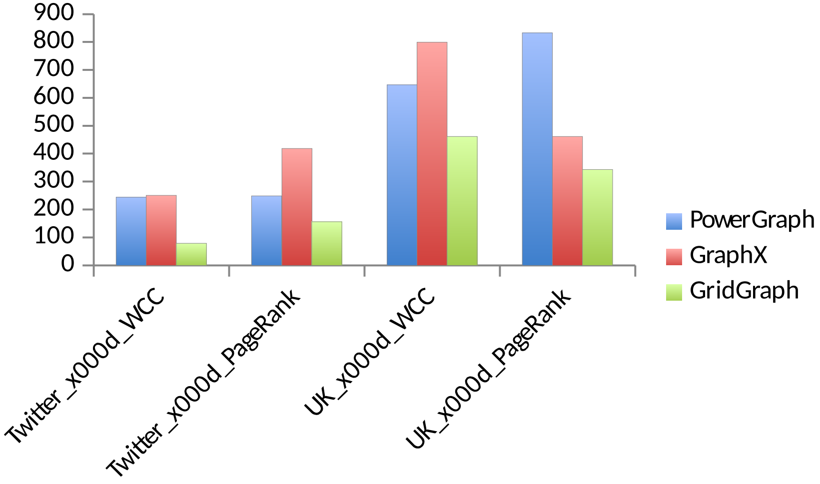
Impact of Q on Out-of-Core Performance



SpMV on Yahoo
Memory limited to 8GB
Optimal at Q=2 where needed vertex data can be fit into memory.

Comparison with Distributed Systems

- PowerGraph, GraphX *
  - 16 × m2.4xlarge, $15.98/h
- GridGraph
  - i2.4xlarge, $3.41/h
    - 4 SSDs
    - 1.8GB/s disk bandwidth



* GraphX: Graph Processing in a Distributed Dataflow Framework, JE Gonzalez et al., OSDI 2014

Conclusion

- GridGraph
  - Dual sliding windows
    - Reduce I/O amount, especially writes
  - Selective scheduling
    - Reduce unnecessary I/O
  - 2-level hierarchical grid partitioning
    - Applicable to 3-level (cache-memory-disk) hierarchy

Thanks!