Graph Analytics in Storage

Sang-Woo Jun

Guest lecture for 6.886 (Graph Analytics)

2018-03-02

Size of Graphs in Nature

	Vertices	Edges
Road Network	10 Millions	100 Millions
Social Networks	Billions	10 Billions
Web Graphs	10 Billions	100 Billions
Brain Neural Network	100 Billions	Trillions

Just a general scale!

Example Open Dataset: Web Data Commons Web Graph

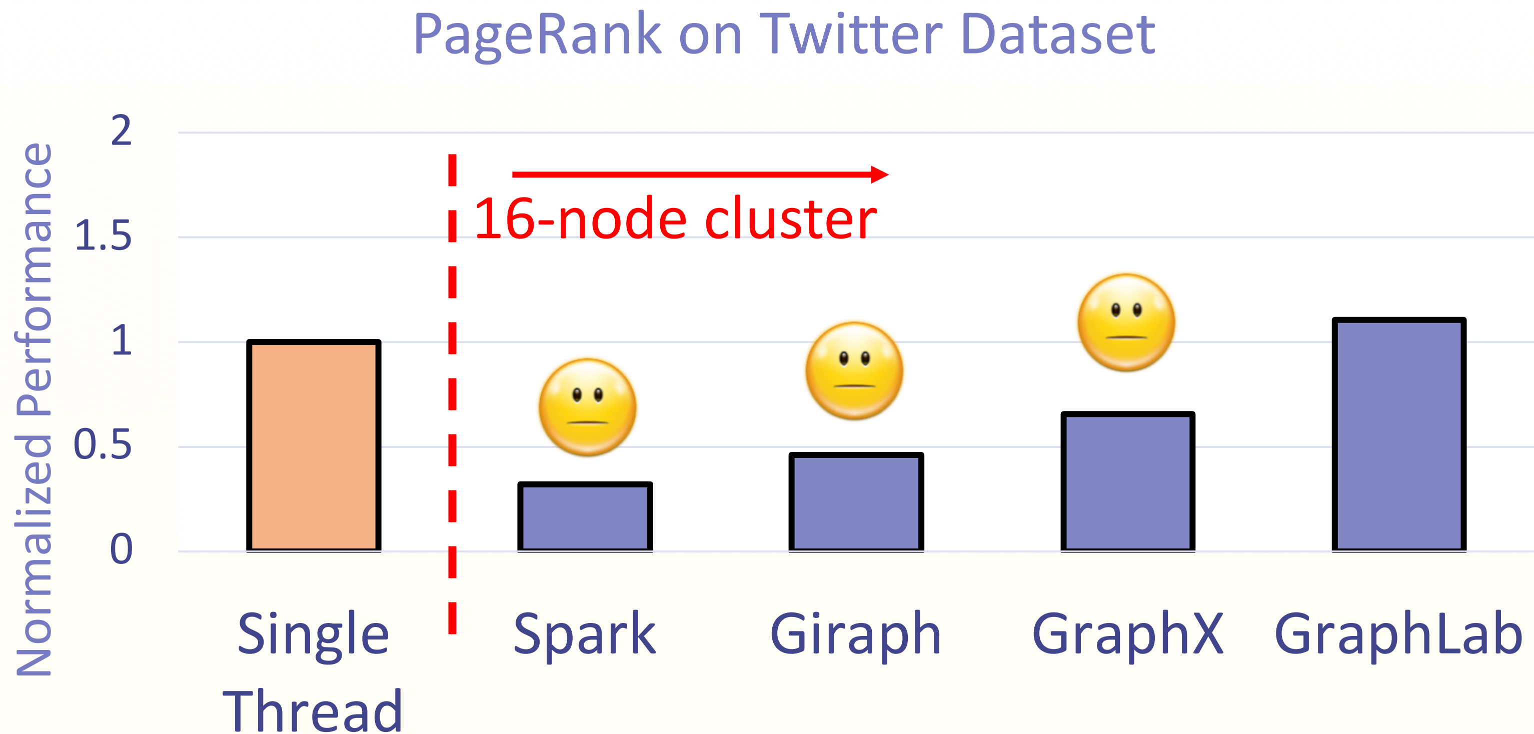
- Hyperlink graph collected by Common Crawl
- “[…] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft.”
- 3.5 billion web pages and 128 billion hyperlinks
- 2 TB in text (0.5 TB encoded)

Compare against the Twitter dataset
40 Million vertices, 30 GB

Machines of Scale

- An $8,000 machine
  - 32 Cores
  - 128 GB DRAM
- Cost of Scale-Out
  - 8+ Machines for 1TB DRAM
  - DRAM also used by OS, FS, Disk cache, network buffer…
  - $64,000 in machine cost + Network infrastructure + …

Scale-Out Incurs Significant Overhead



“Scalability! But at what COST?,” Frank McSherry, HotOS 2015

Cost of Scale-Up

- TBs of DRAM on a single machine incurs non-linear cost increase
  - Goes into HPC (High-Performance Computing) area
  - Custom designed hardware/architecture
  - Very expensive!
- Can we not use DRAM to handle capacity?
  - (Cheap hard disks for example?)
  - HDD 1 TB costs ~ $50 (SATA)
  - SSD 1 TB costs ~ $500 (PCIe)
  - DRAM 1 TB costs ~ $8,000

Contents

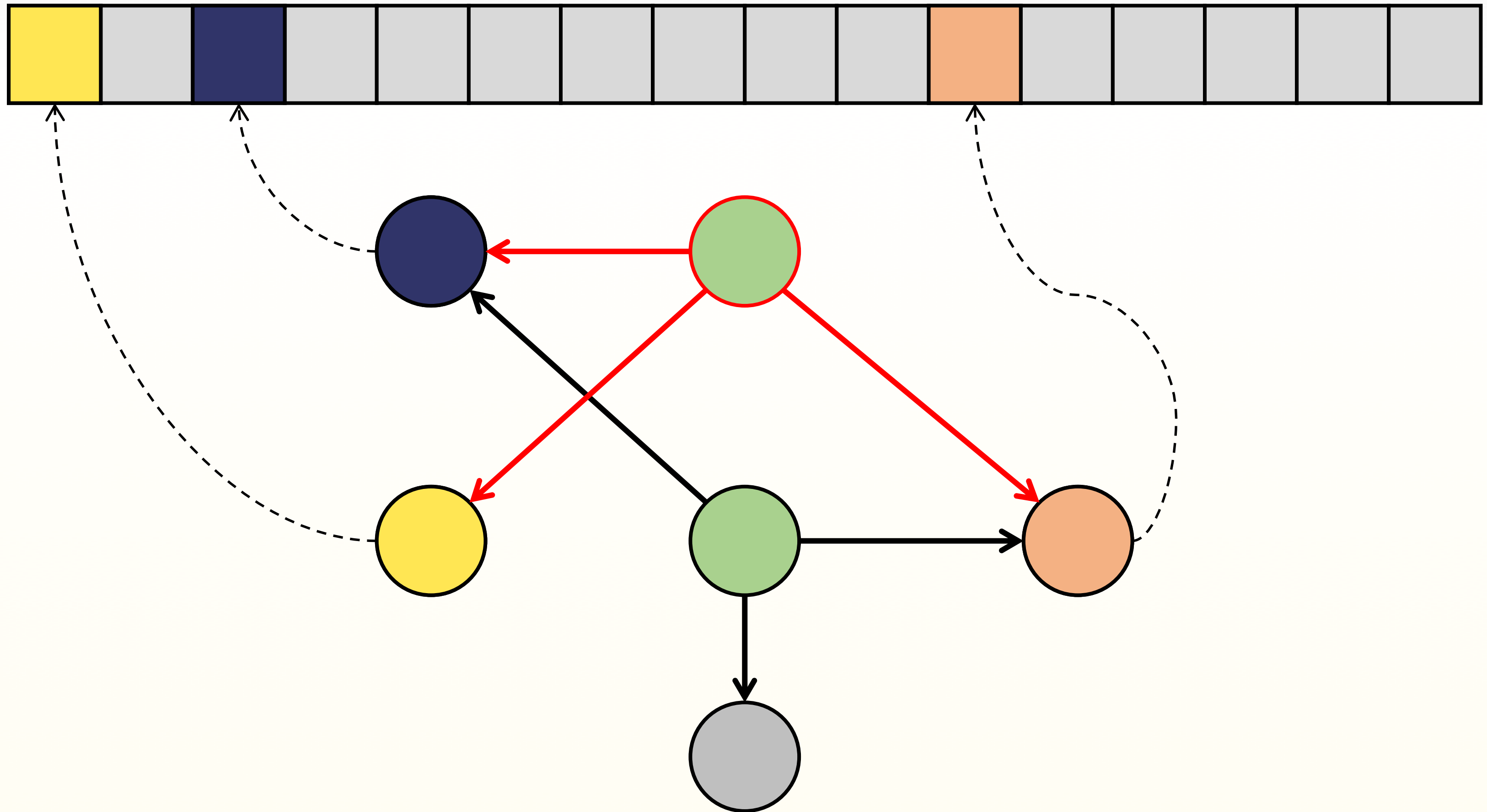
- Why storage is not a good fit for graph analytics
- How some systems overcome these limitations
  - GraphChi
  - FlashGraph
  - Mosaic
  - BigSparse

Characteristics of Large Graphs

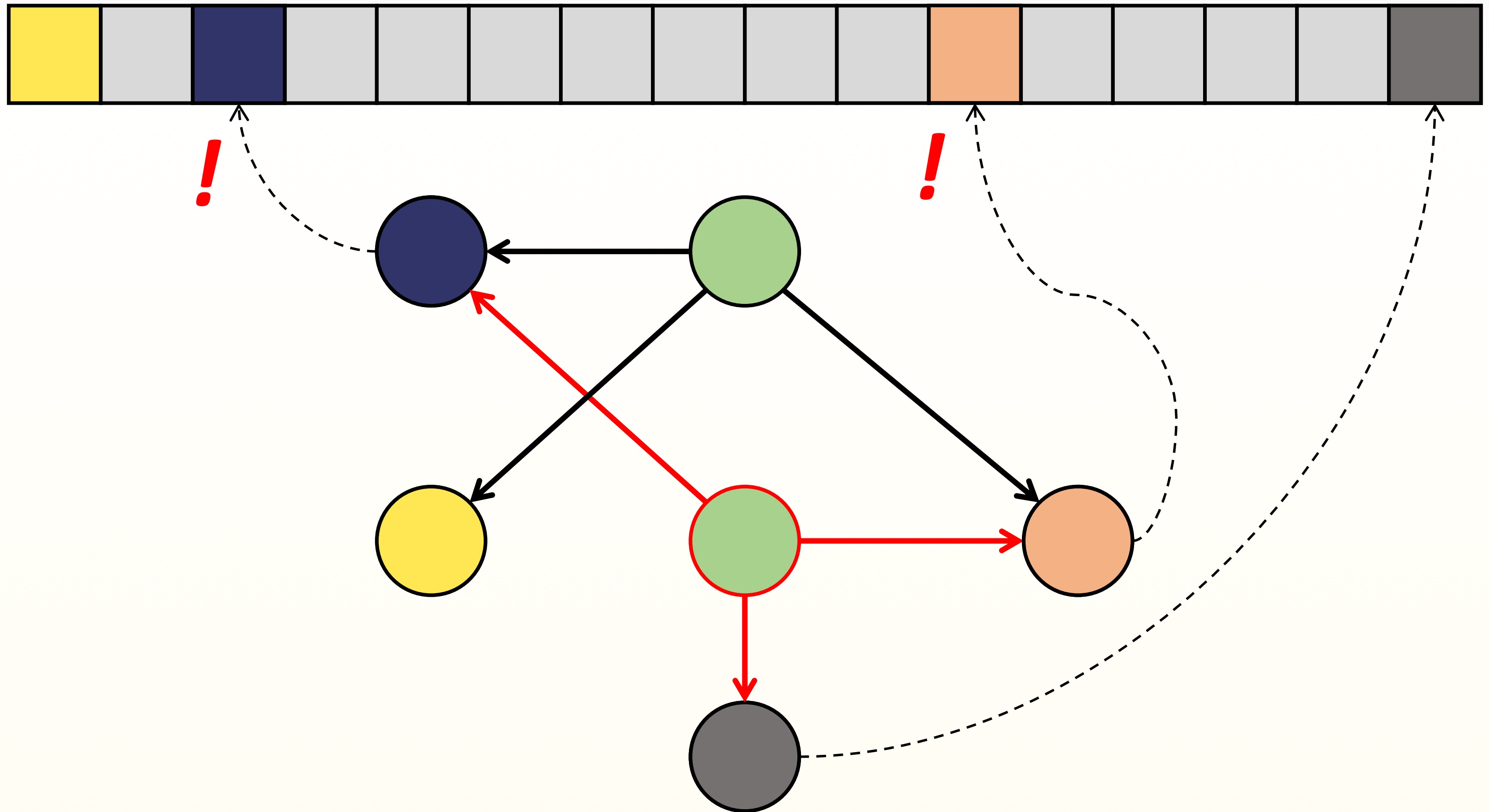
- Large (of course)
  - Multiple TBs
- Sparse
  - Edge factor of 10s or 100s
- Irregular
  - Not much locality
  - Any vertex can be connected to any other vertex

Irregularity + Sparsity → Fine-grained random access!

Random Access Within an Active Vertex



Random Access Across Active Vertices



Data size and irregularity limit cacheing effectiveness

DRAM vs Disk vs SSD

Pay attention to the units! (e.g., GB vs MB)

	DRAM	HDD	SSD
Cost/TB	$8,000	$50	$500
Bandwidth	200 GB/s	750 MB/s	4 GB/s
Latency	20 ns	10 ms	20 us
Power (W)	200	2 – 5	2 – 5

- HDD Bandwidth assumes SATA-III (6Gb/s)
- HDD Latency for random reads
  - High mechanical seek time penalty
- SSD Bandwidth assumes PCIe Gen2 x8 (4GB/s)

For performance, storage reads must be in coarse granularities (Megabytes for HDD, Kilobytes for SSD)

Issue of Access Granularity

- Minimum unit of storage access is a page (4 – 8KB)
  - Reading 8 KB to use 8 bytes is a waste (1/1024 bandwidth)
- Minimum unit of DRAM is complicated
  - 128 Byte cache line?
  - 1 – 8KB row buffer?
  - But DRAM has much lower latency

For performance, storage reads must be in coarse granularities (Megabytes for HDD, Kilobytes for SSD)

AND we must organize data so that most data read is useful

Software Interface for Storage Access

- Typically using blocking read() operations
  - Blocking random access kills performance
  - Remember ***10 ms*** vs ***20 us*** vs ***10 ns*** difference!
  - Per thread SSD: 100MB/s HDD: < 1MB/s
- Asynchronous I/O using more threads
  - Lots of threads doing blocking reads
  - 40+ Threads to reach SSD bandwidth (RocksDB)
- Linux Kernel Asynchronous I/O
  - Please let me know if you can get it to perform!

Contents

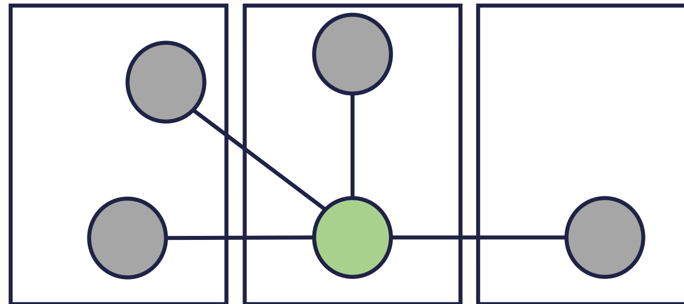
- Why storage is not a good fit for graph analytics
- How some systems overcome these limitations
  - GraphChi
  - FlashGraph
  - Mosaic
  - BigSparse

GraphChi: Large-Scale Graph Computation on Just a PC

- The first graph analytics system in storage
  - Based on observations from GraphLab
- Novelty: Parallel Sliding Window algorithm
  - Can function on systems with very small memory (MBs)
  - Optimized for reducing random memory access

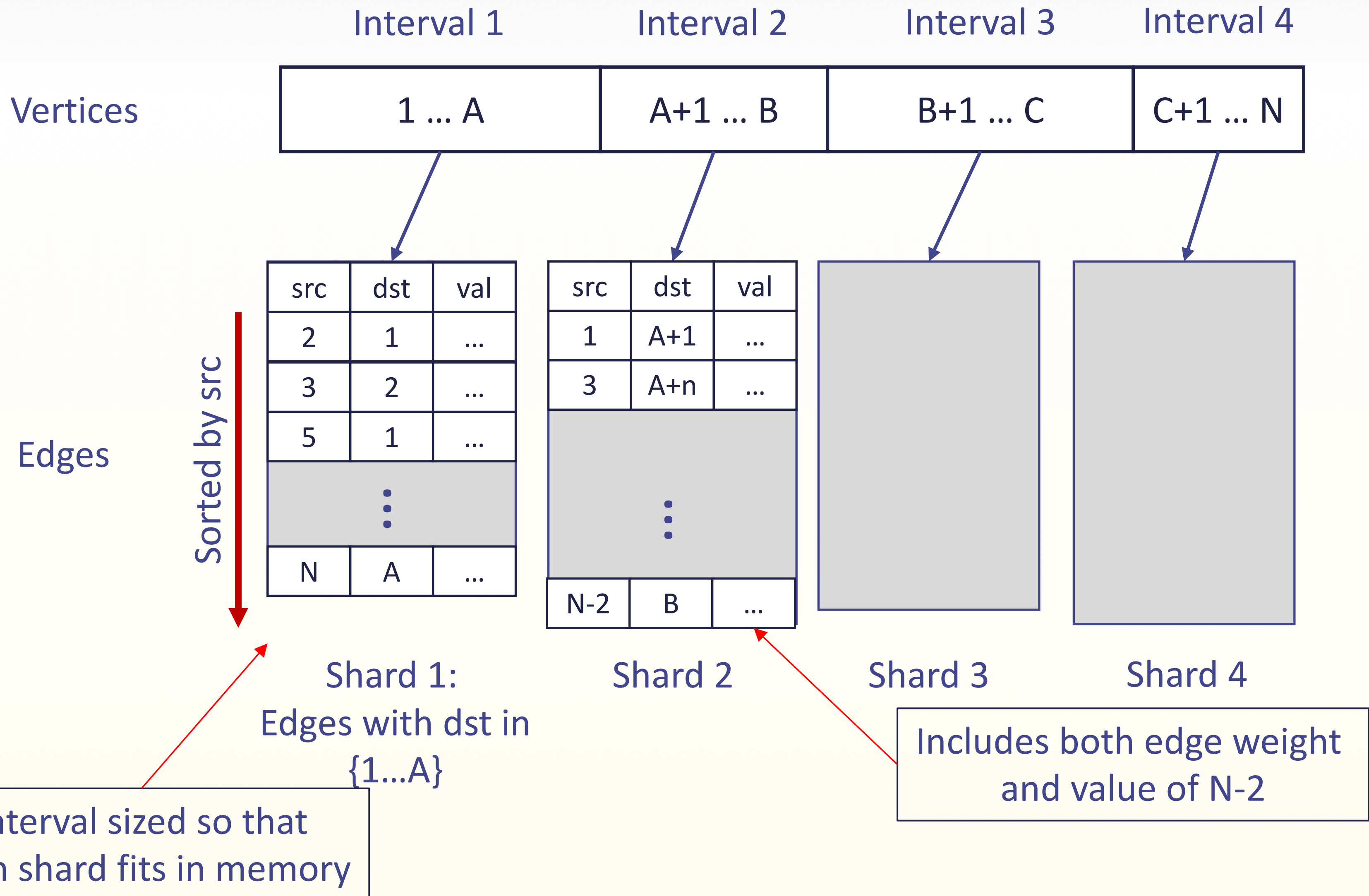
Parallel Sliding Window – Motivations

- Hurdles of partitioning
  - Process in-edges: random read across vertex partitions
  - Process out-edges: random write across vertex partitions



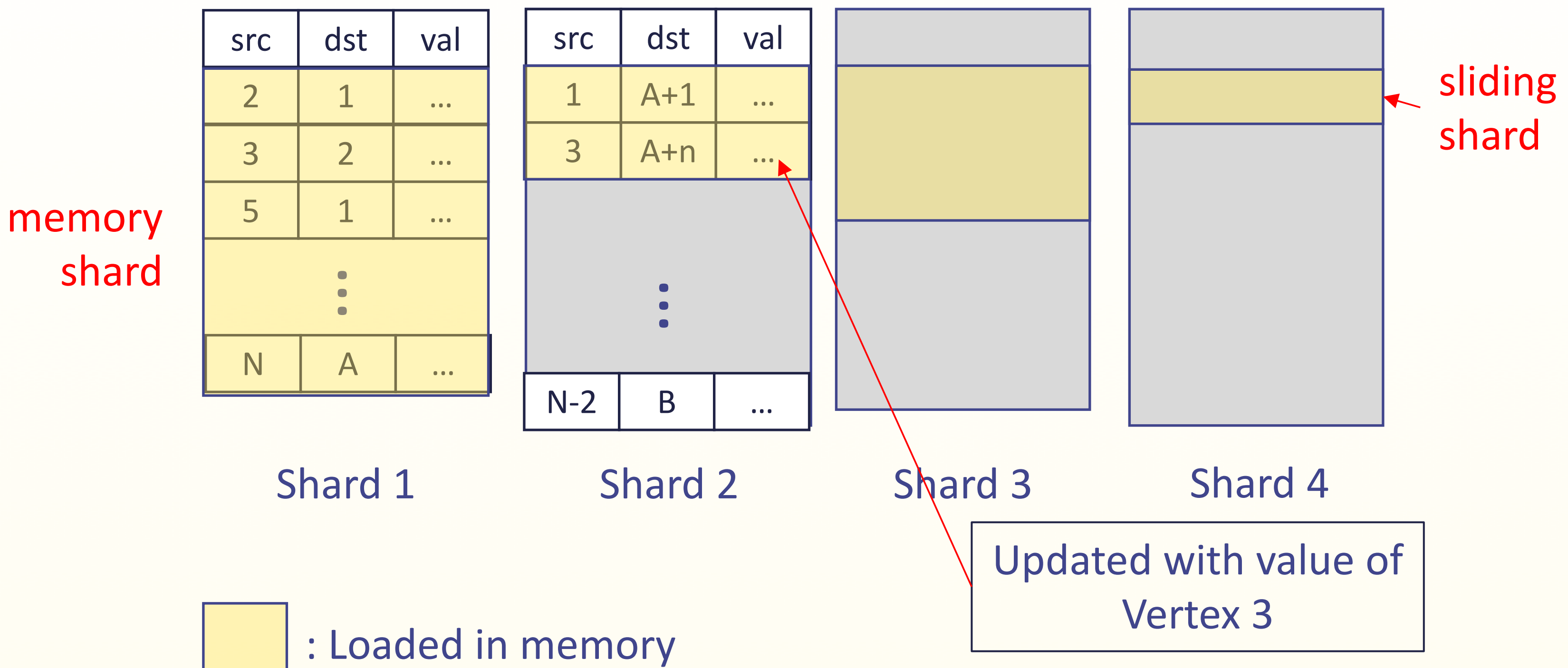
- Parallel Sliding Window's solution
  - Collocate vertex data with edge data
  - "Send source vertex's values to neighbors"
  - Some duplication of data

Parallel Sliding Window - Partitions



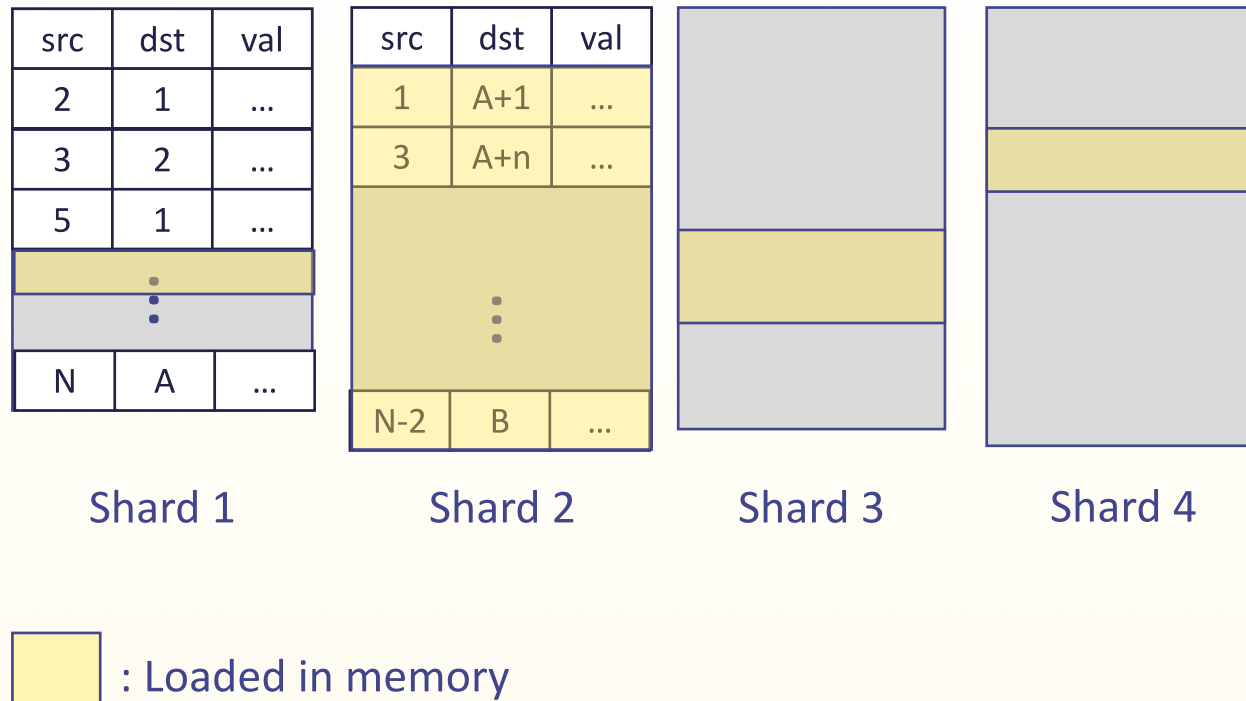
Parallel Sliding Window - Execution

- Algorithm iterates over intervals
- For interval 1, only load shard 1 and parts of shards that have src in interval 1



Parallel Sliding Window - Execution

- Next shard is loaded in memory
- Sliding shards move forward to match next shard



Selective Scheduling in GraphChi

- For algorithms with sparse active set (e.g., Breadth-First-Search), inefficient to process all edges
- GraphChi's method: coarse-grained selection
  - Divides each shard into sub-indices
  - When neighbors are activated, a bit mask is set
  - Loops through the bitmask to determine which sub-indices to skip
  - … I think that's what it says it's doing

Benefits of PSW

- Most reads are sequential chunks
- For P shards, only P^2 random jumps in reads
  - Across sliding shard reads
- Each edge is read up to two times
- Each edge is written up to two times

Shortcomings of PSW

- Initial preprocessing (sharding) overhead is high
  - 10 mins to load twitter graph
- Vertex value is duplicated
- Selective scheduling is inefficient
  - Coarse granularity?
  - Loop through bit mask?
  - Results with selective scheduling is not included in paper

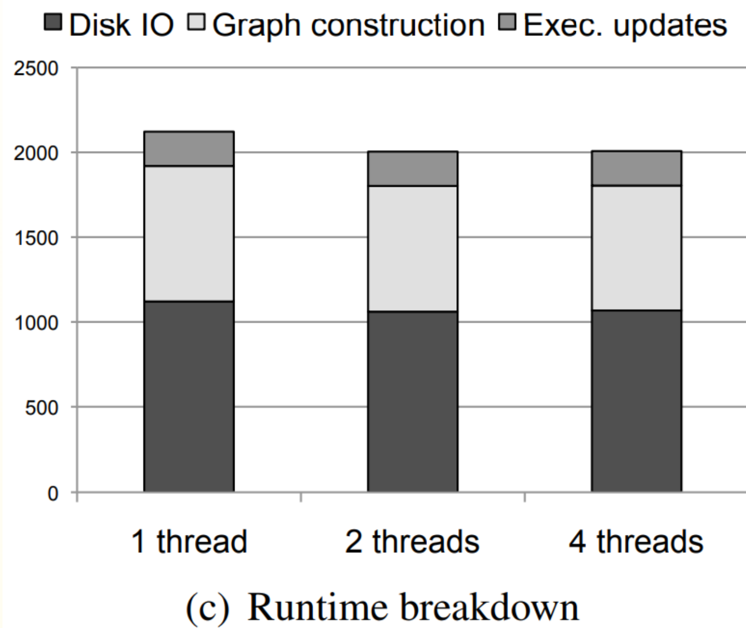
GraphChi Performance Results

- Paper compares against inconsistent system configurations
- Compared to Hadoop-based Pegasus
  - Similar to Pegasus on 100 machines
- Compared to in-memory systems
  - Half performance against single-node GraphLab
  - Half performance against 50-node Spark

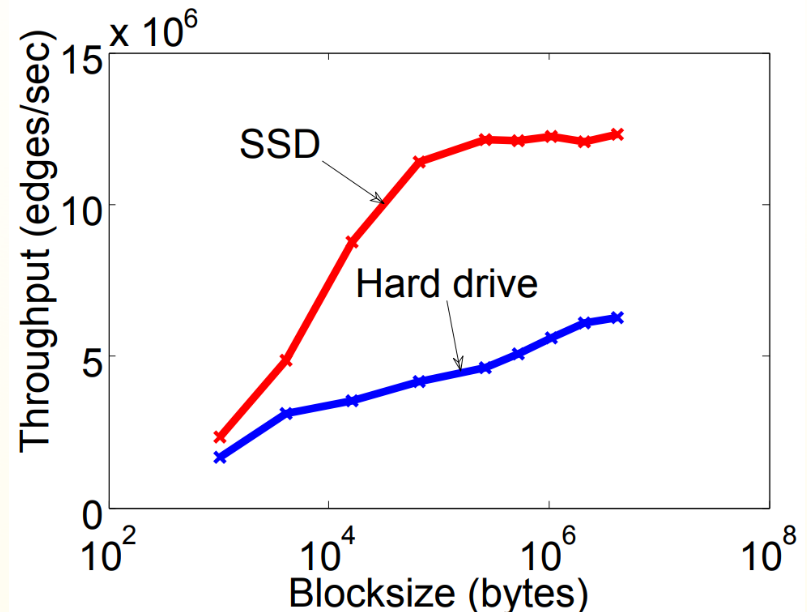
More consistent results will be presented later

Configuration Impact on Performance

- Linear performance scaling with more disks
- Multithreading does not buy much performance
- Significant performance improvements by larger blocks



(c) Runtime breakdown



Contents

- Why storage is not a good fit for graph analytics
- How some systems overcome these limitations
  - GraphChi ✓
  - FlashGraph
  - Mosaic
  - BigSparse

FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

- Stores vertex data in memory
- Efficient access to edge data using special file system

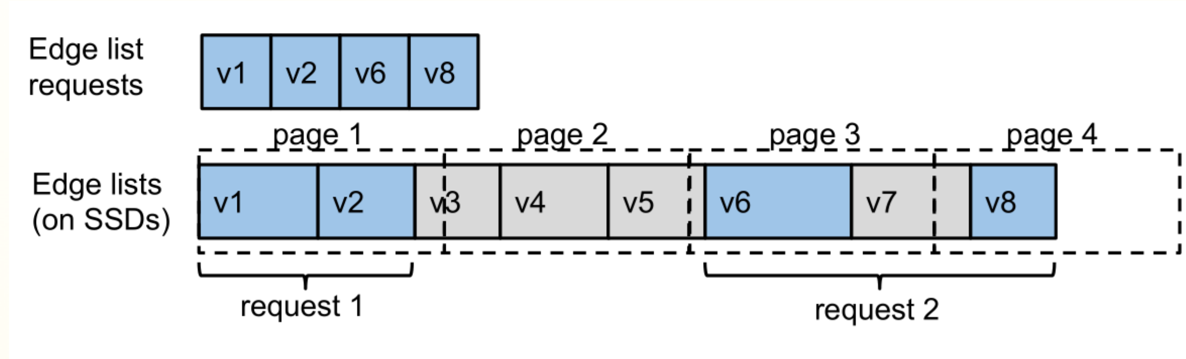
Edge Data vs Vertex Data in Graphs

	Edge Data	Vertex Data
Size:	$O(N \times EdgeFactor)$	$O(N)$
Locality:	In-edges to a vertex are grouped together	Each vertex is independant
Pattern:	Monotonically Increasing Reads	Random Read-Modify-Write

Storing vertex data in memory removes a lot of random access

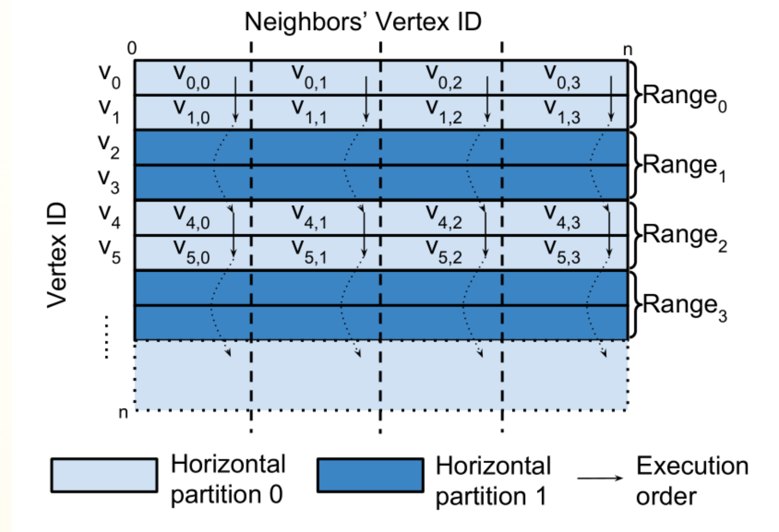
SAFS – Set-Associative File System

- Spawns I/O threads to provide application with asynchronous file I/O
- Dynamically merges SSD access to better use bandwidth



Graph Partitioning

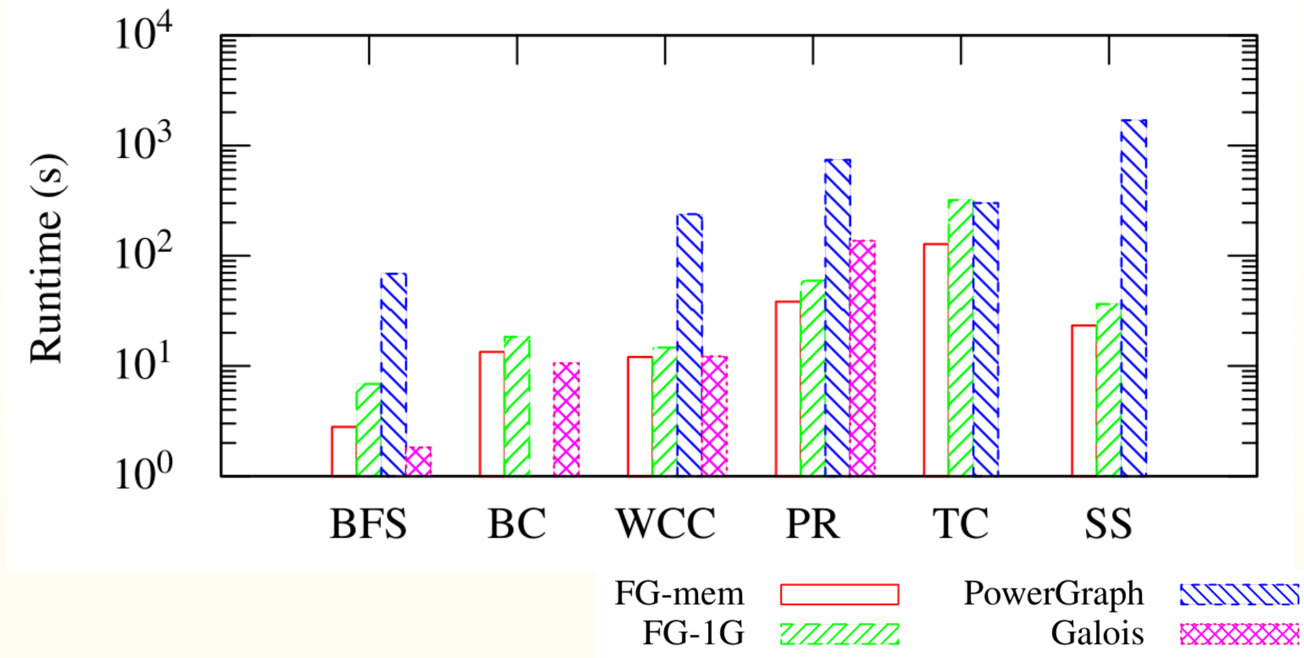
- ❑ Partitions are striped for better balancing
- ❑ Both horizontal and vertical partitioning
  - Horizontal – Partition across vertices
  - Vertical – Partition across neighbors
- ❑ Inter-partition messages batched by threads
- ❑ Inter-thread work stealing



Simple to do thanks to vertex data in memory

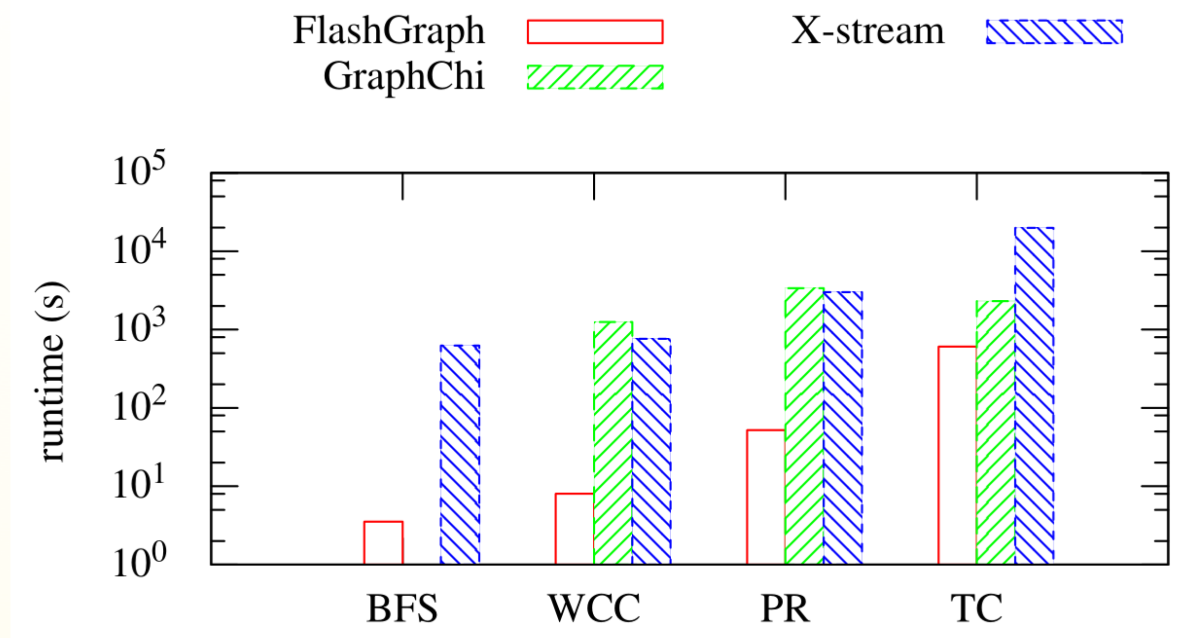
FlashGraph Performance vs In-Memory

- Performance on the Web Data Commons graph
- Comparable storage while loading from flash
- At high IOPS of flash, CPU runs out before flash bandwidth



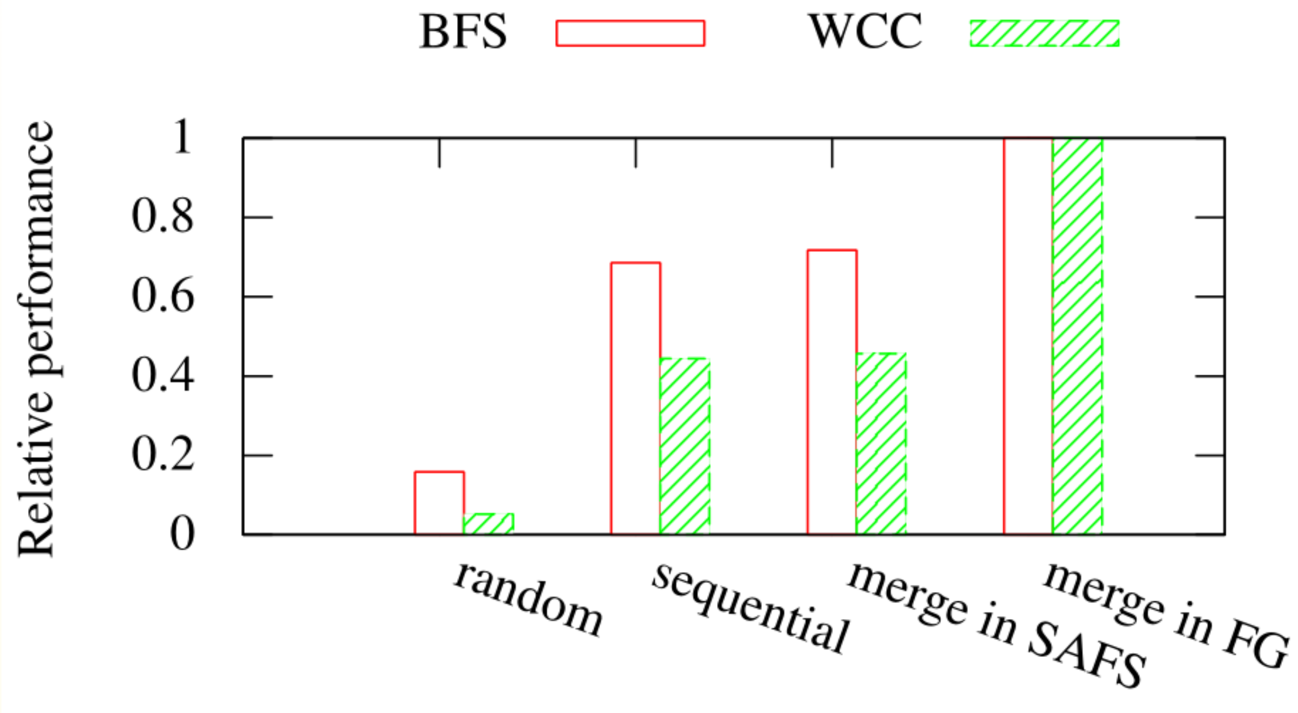
FlashGraph Performance vs External

- Performance on the twitter graph
- Much faster than GraphChi



Performance Impact of Merging Edge Access

- Normalized to merging in FlashGraph
- Significant improvement!



Contents

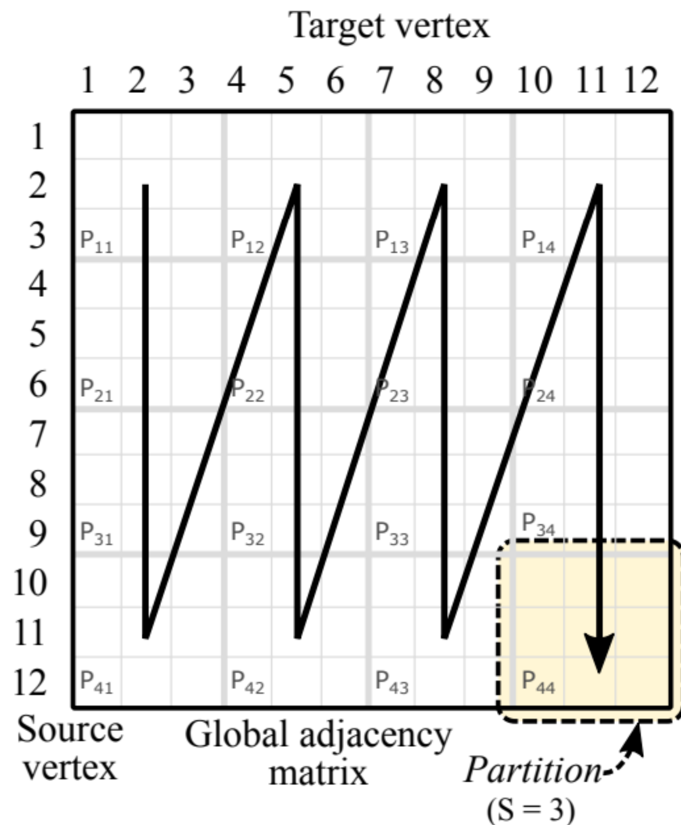
- Why storage is not a good fit for graph analytics
- How some systems overcome these limitations
  - GraphChi ✓
  - FlashGraph✓
  - Mosaic
  - BigSparse

Mosaic: Processing a Trillion-Edge Graph on a Single Machine

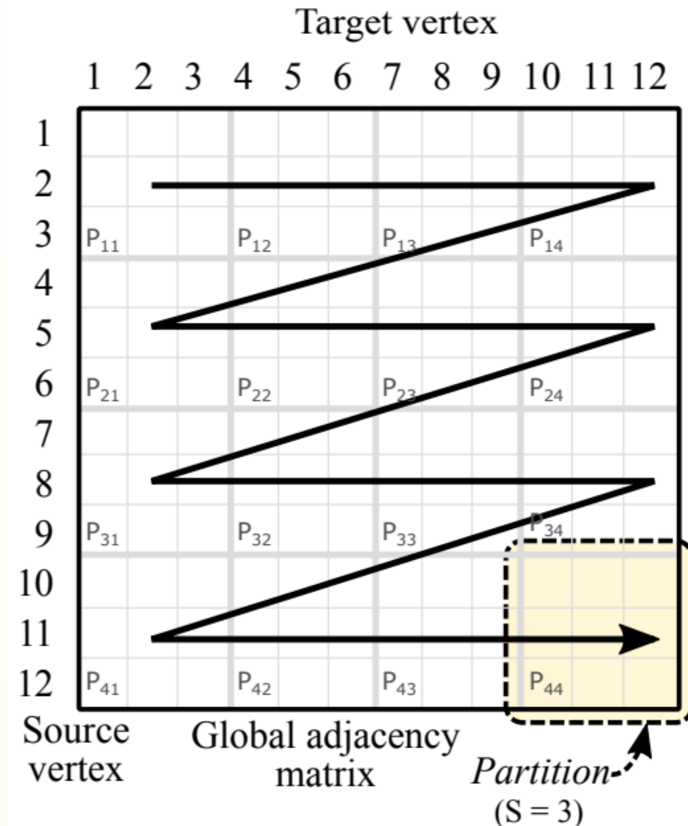
- Like FlashGraph, stores only vertices in memory
- Hilbert order tiles organization to improve locality
- Xeon Phi coprocessor
  - Parallelize SSD access
  - Parallelize edge processing
  - Parallelize vertex processing

Background – Graph Representation

Column Major
Locality for write
Repeated reads

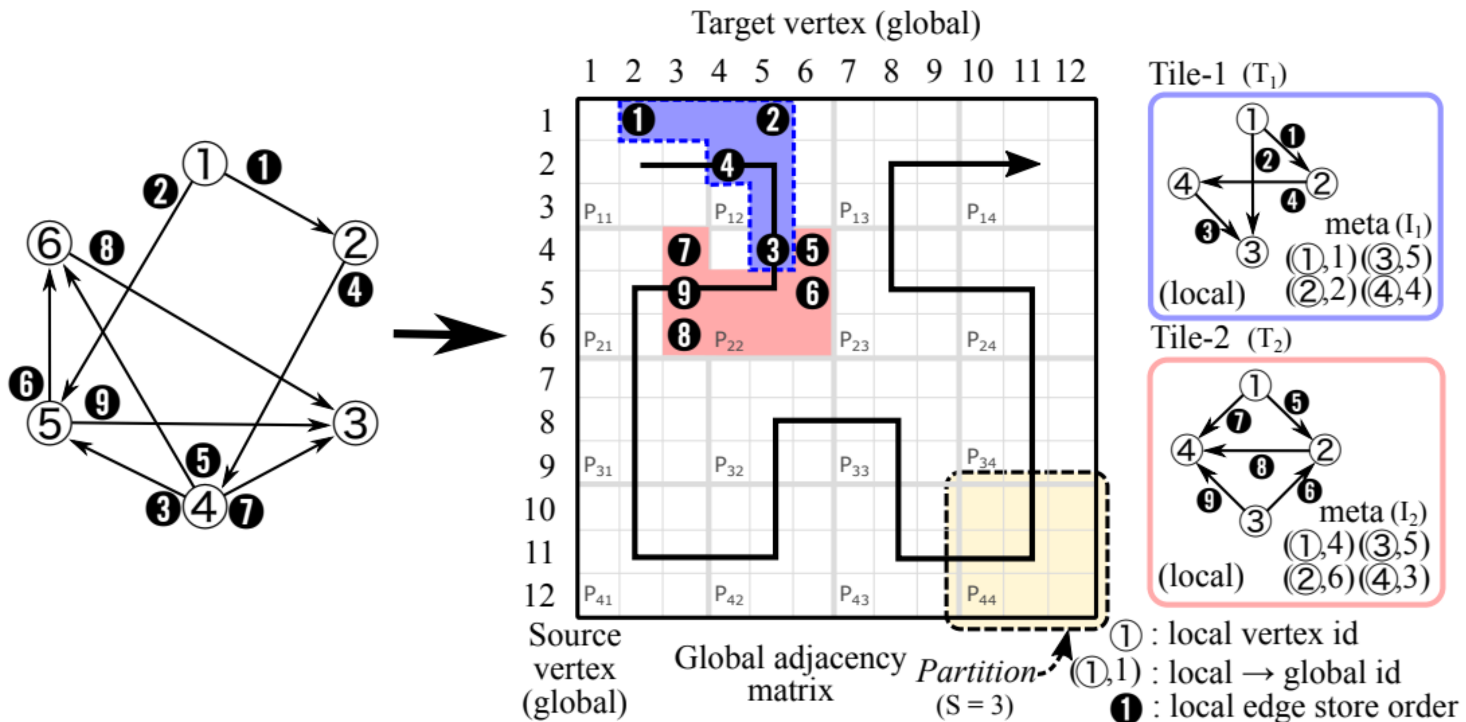


Row Major
Locality for read
Repeated writes



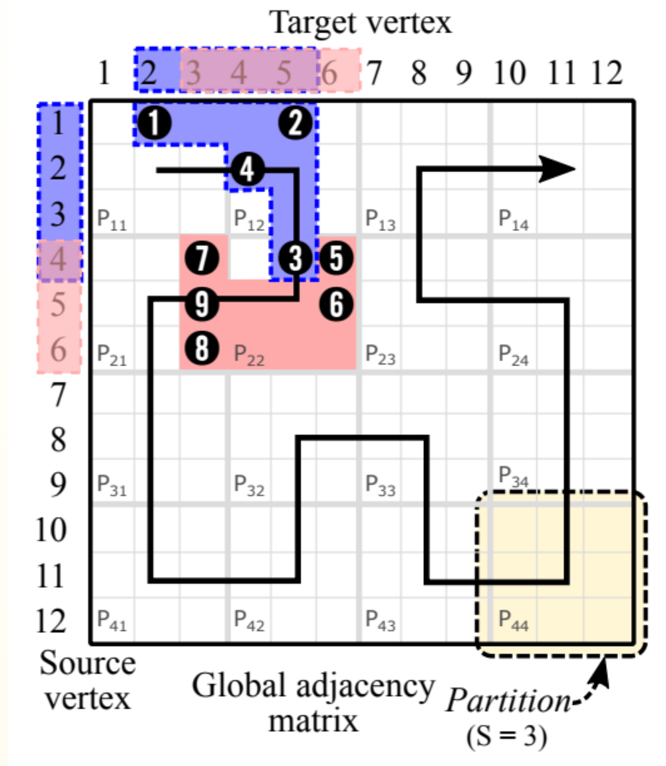
Hilbert-Ordered Tiles

- ❑ Hilbert curve – Fractal space-filling curve
- ❑ Traverses tiles by Hilbert order
  - Until reaching vertex count limit per tile



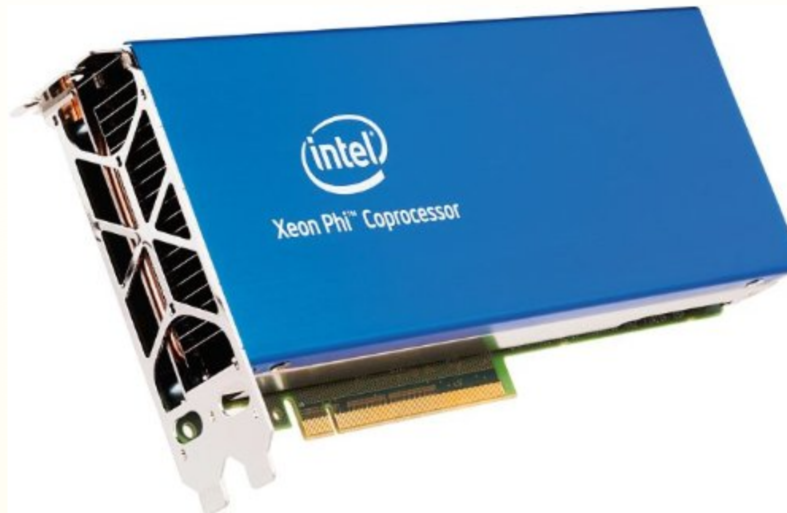
Hilbert Order Has Good Locality

- Both sources and targets have overlap



Xeon Phi Coprocessor

- Intel’s answer to GPU accelerators
- 64-72 x86 cores
  - With Intel SIMD instructions
- Hundreds of GB of memory



Use of Xeon Phi in Mosaic

- Each Xeon Phi core sends read requests directly to NVMe via DMA
  - Many many requests in flight
- Each Hilbert order tile fits in Xeon Phi core's LLC
  - Pull and Intra-Tile Reduce performed on Xeon Phi core
- Inter-Tile Reduce performed in host server
  - Intra-inter tile reduction separation made possible by associative nature of Reduce

Pull-Reduce-Apply Model

- Vertex Program is divided into three parts
- Pull (Vertex src, Vertex dst)
  - Gather per edge information
  - Uses incoming neighbor value and current local vertex
- Reduce (Vertex v1, Vertex v2)
  - Given two incoming edges, reduce into one
  - Must be associative
- Apply (Vertex v)
  - Calculate non-associative math

Pull-Reduce-Apply Example

Edge-centric operation

Local graph processing on Tile

```
1  // On edge processor (co-processor)
2  // Edge e = (Vertex src, Vertex tgt)
3  def Pull(Vertex src, Vertex tgt):
4      return src.val / src.out_degree
5  // On edge processor/global reducers (both)
6  def Reduce(Vertex v1, Vertex v2):
7      return v1.val + v2.val
8  // On global reducers (host)
9  def Apply(Vertex v):
10     v.val = (1 - α) + α × v.val
```

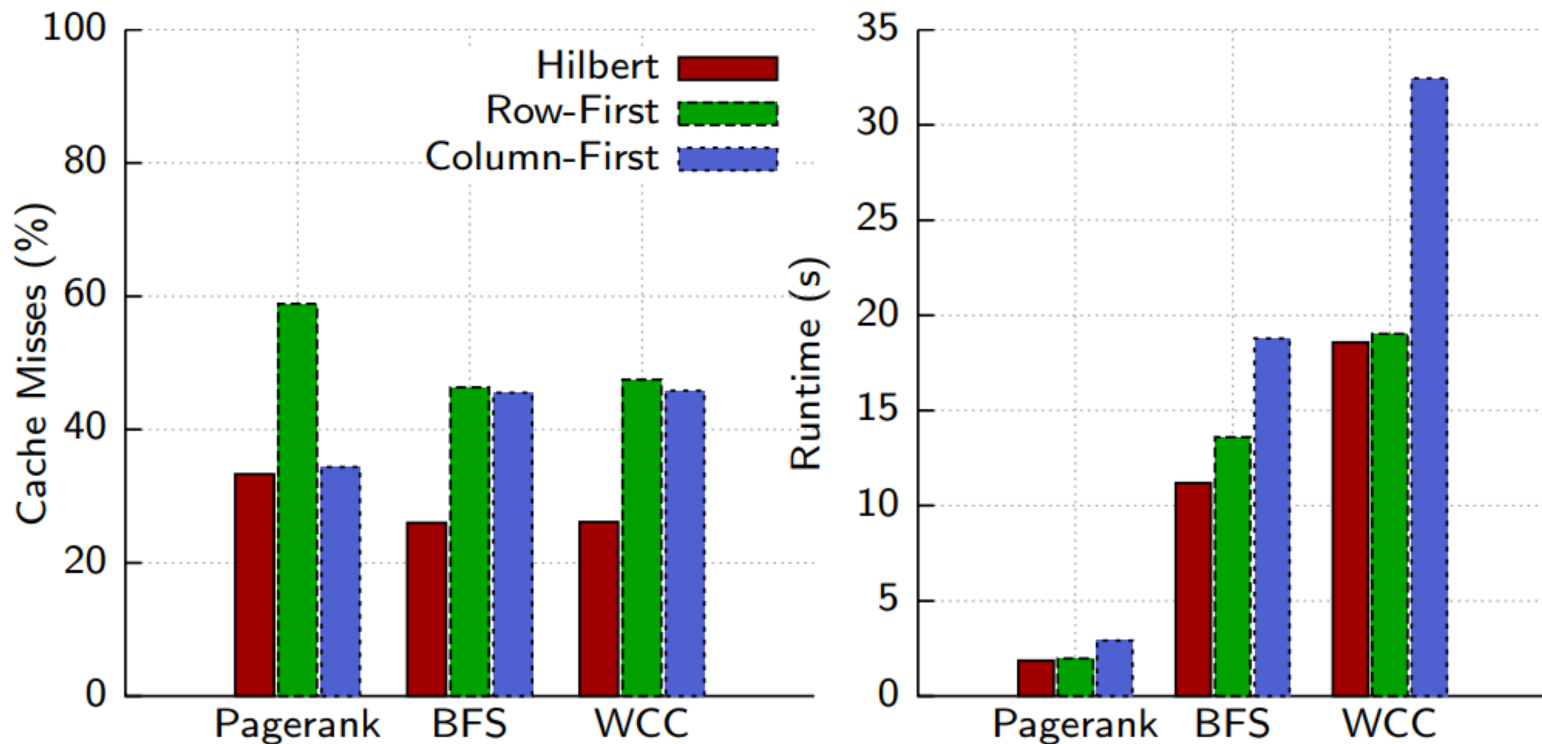
Global graph processing

Vertex-centric operation

$$Pagerank_v = \alpha * \left(\sum_{u \in Neighborhood(v)} \frac{Pagerank_u}{degree_u} \right) + (1 - \alpha)$$

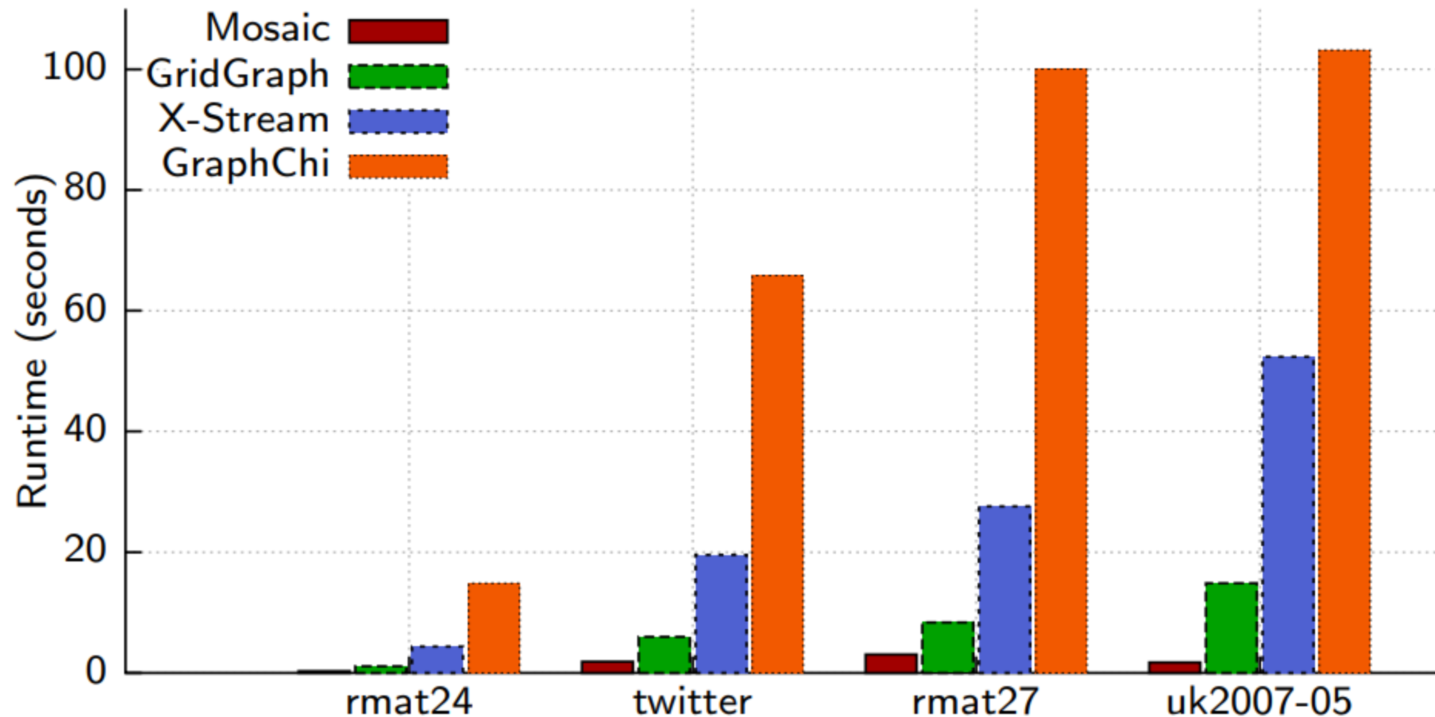
Performance Benefits of Hilbert Ordering

- Increased locality translates to performance



Mosaic Performance Against Storage-Based Systems

- Much better compared to other storage-based system
  - Compared systems don't use Xeon Phis



Comparisons Against More Systems

- Against In-Memory Systems
  - Comparable performance against Polymer and Ligra
  - 1.8x slower than Polymer
  - 2x faster than Ligra
- Against GPU-accelerated systems
  - Slower compared to TOTEM and GTS
  - 3.3x slower than TOTEM
  - 2.6x – 1.4x slower than GTS

Contents

- ❑ Why storage is not a good fit for graph analytics
- ❑ How some systems overcome these limitations
  - GraphChi ✓
  - FlashGraph ✓
  - Mosaic ✓
  - BigSparse

BigSparse: High-performance external graph analytics

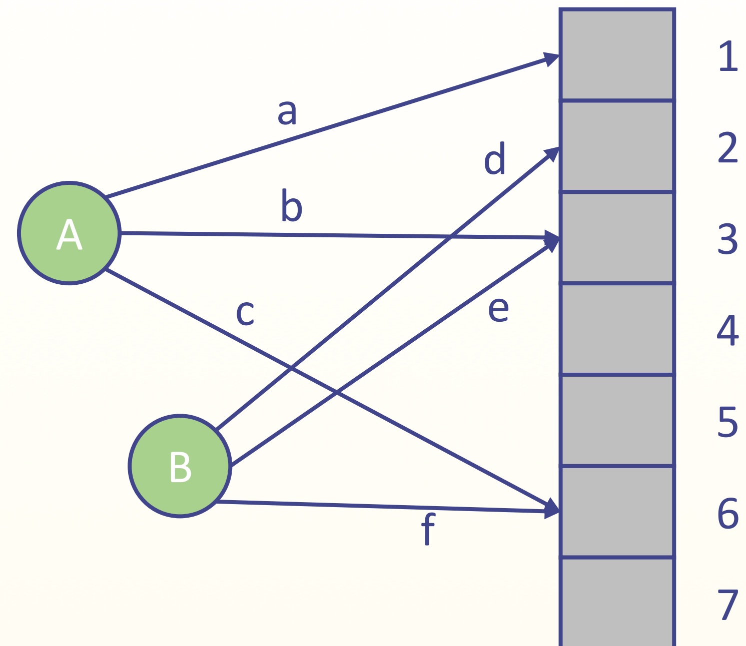
- Fully external analytics
  - Even vertex data in storage
  - Very little memory required
- Novelty: Sort-Reduce algorithm to sequentialize storage updates

Random Access in Push-Style Vertex Program

- Update operations in a Shortest Path Example

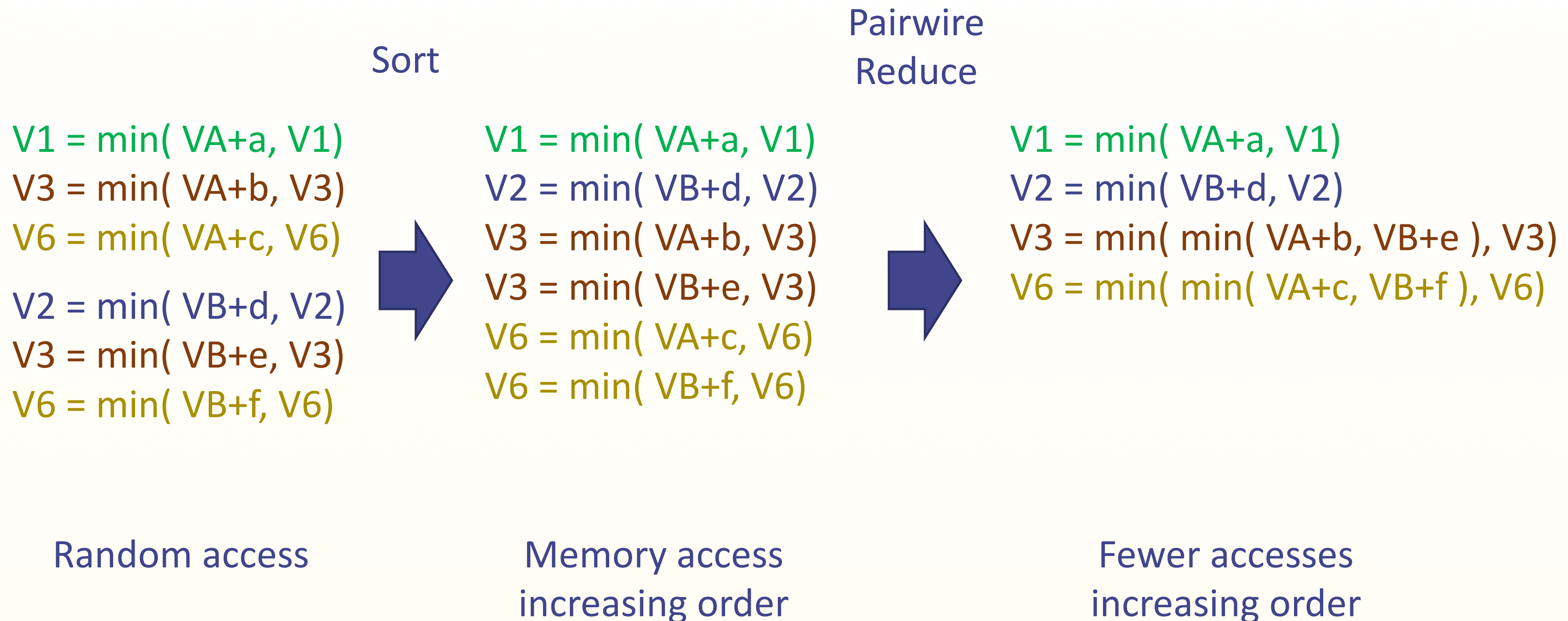
V1 = min(VA+a, V1)
V3 = min(VA+b, V3)
V6 = min(VA+c, V6)

V2 = min(VB+d, V2)
V3 = min(VB+e, V3)
V6 = min(VB+f, V6)



Fine-granularity random read/writes

Better Organization of Accesses: Sort-Reduce

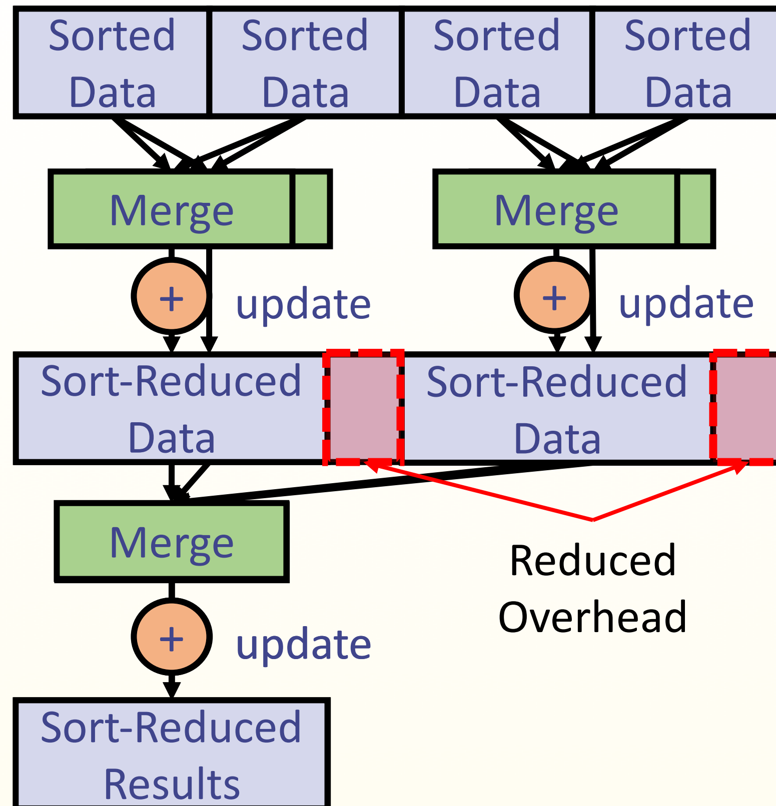


Thanks to associative reductions

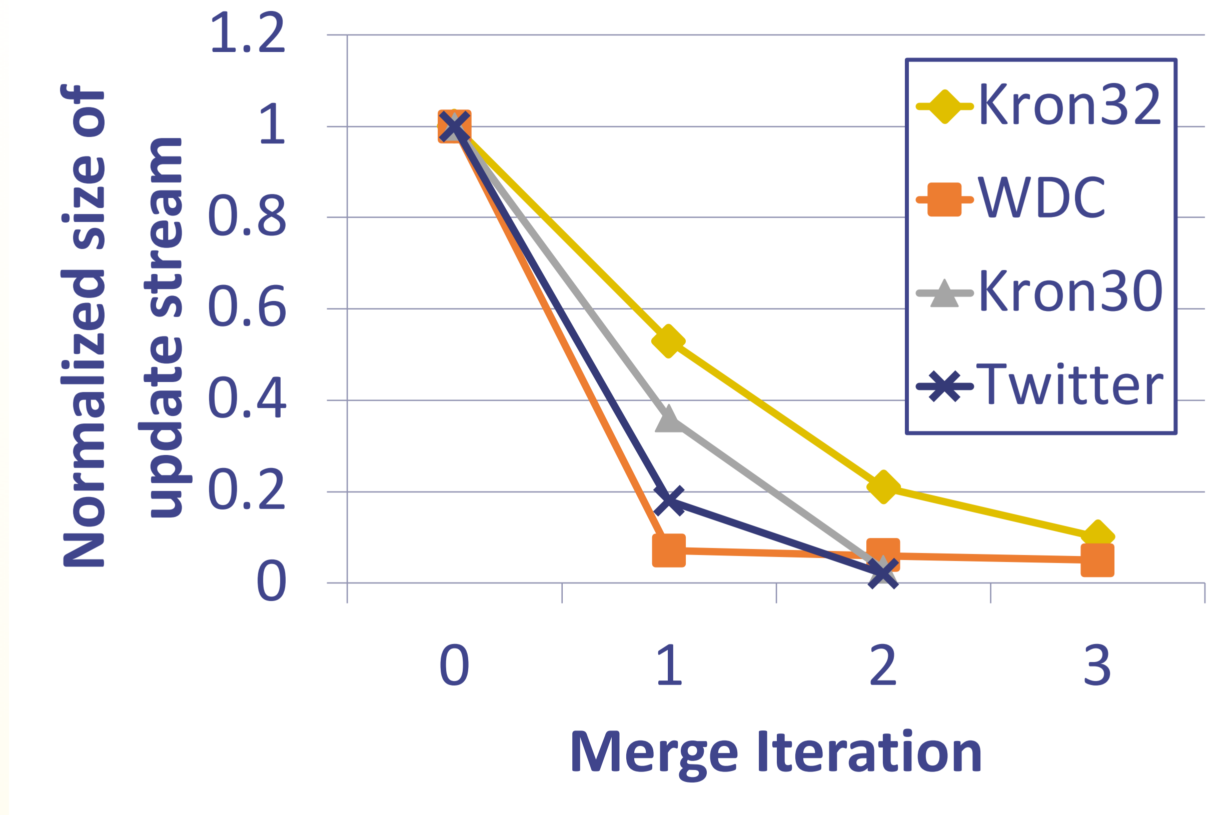
Putting it Together - Sort Reduce

Updates are first logged and sorted using external merge sort

Reduction can be applied after every merge iteration

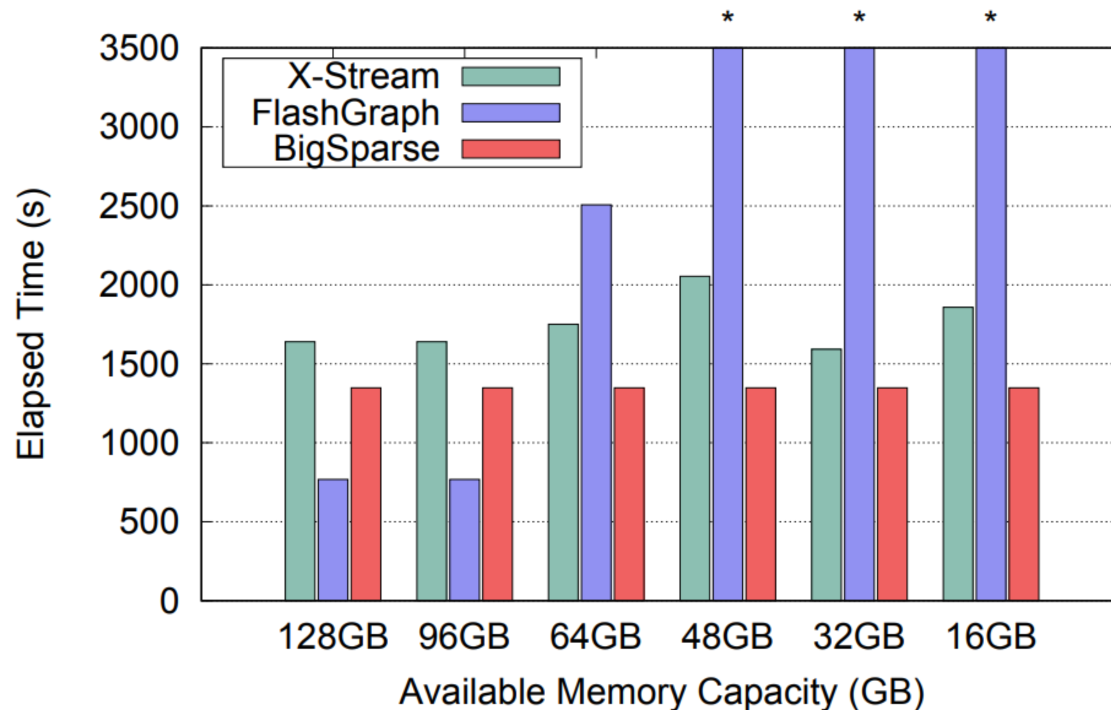


Big Benefits from Interleaving Reduction



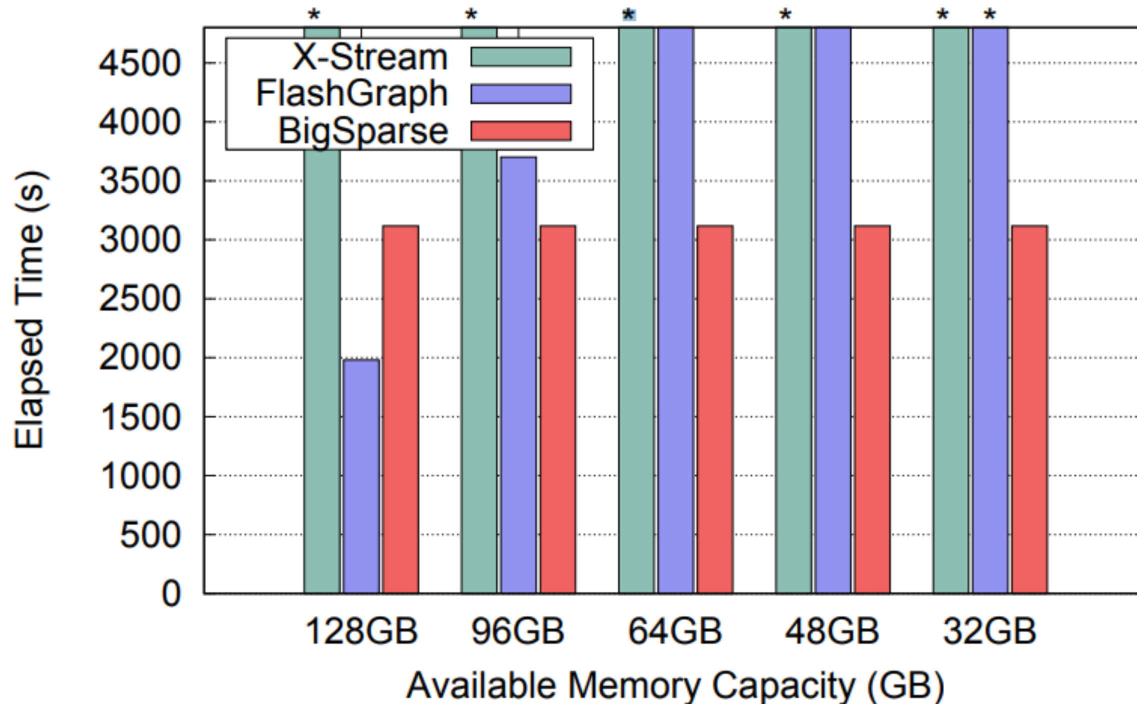
BigSparse Performance Results

- PageRank on the Web Data Commons graph
- FlashGraph starts thrashing at 64 GB memory capacity



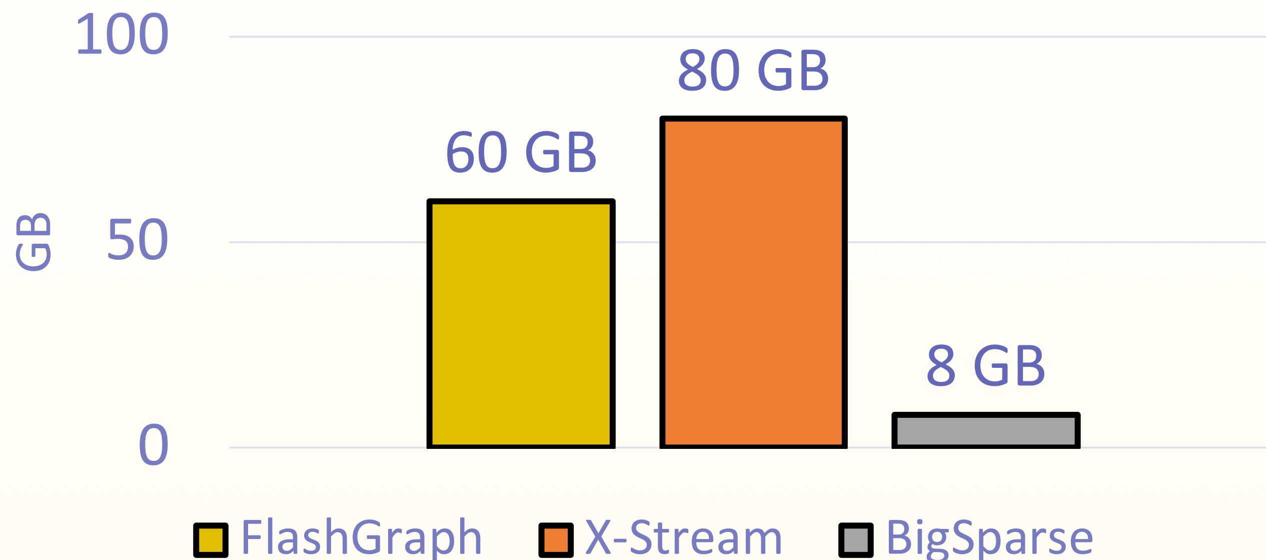
BigSparse Performance Results

- Betweenness-Centrality on the Web Data Commons graph
- FlashGraph starts thrashing at 96 GB memory capacity



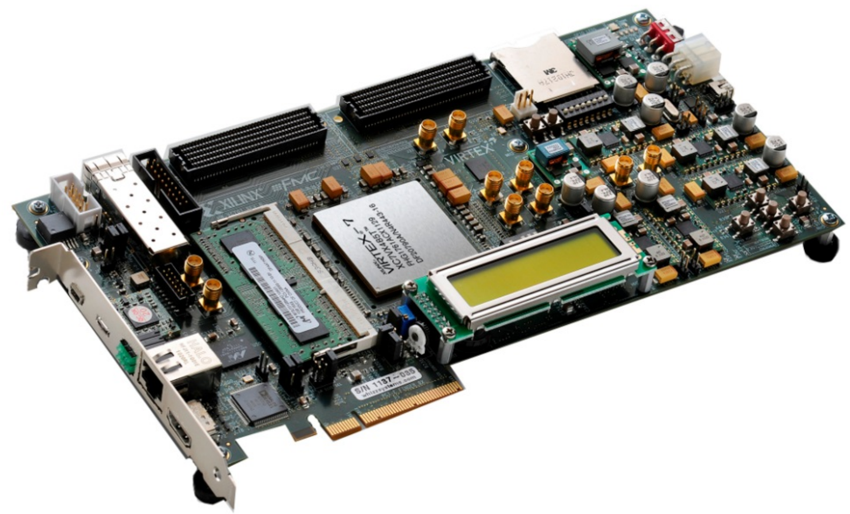
External Analytics Dramatically Decreases Memory Usage

Most of GraFSoft memory usage is flash prefetch buffers



Hardware Sorting Accelerator

- Hardware Sorting Accelerator using Field-Programmable Gate Array (FPGA)
  - Creates dedicated hardware in FPGA chip
  - Low power, high performance
- Performs 4x compared to 8-thread software
  - Can always instantiate more



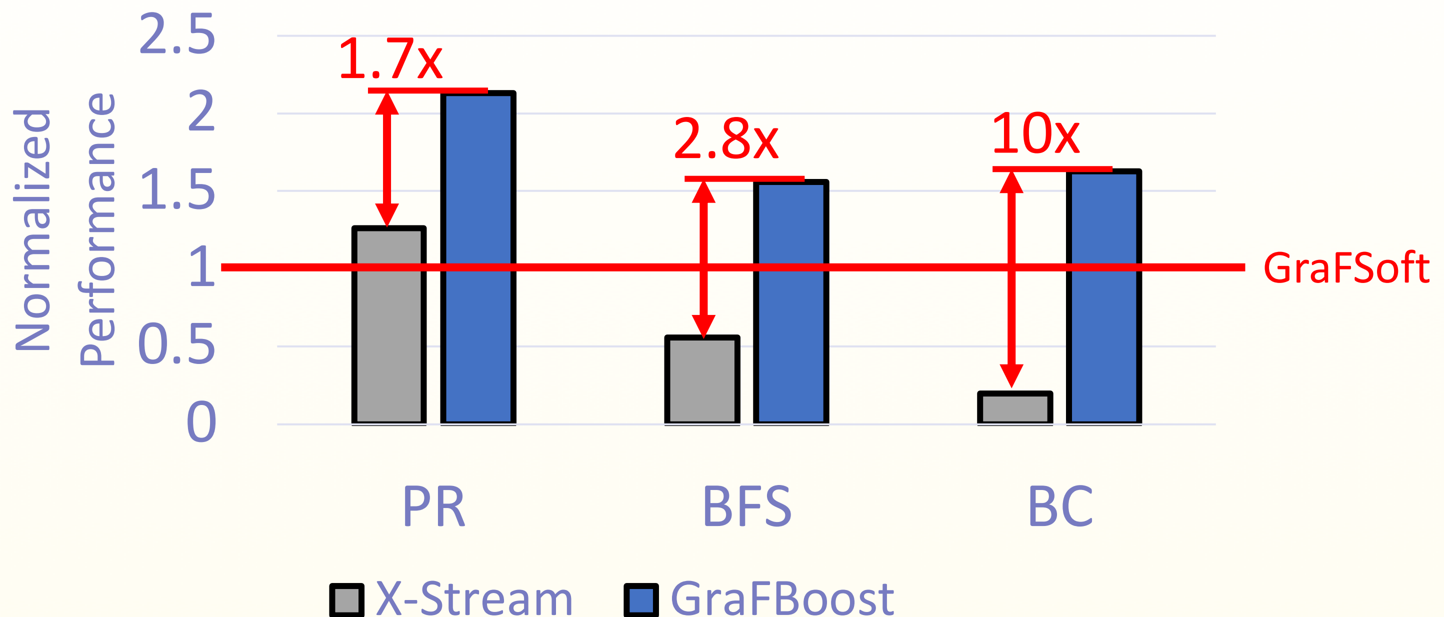
Results with a Large Graph: Synthetic Scale 32 Kronecker Graph

0.5 TB in text, 4 Billion vertices

GraphLab out of memory

FlashGraph out of memory

GraphChi did not finish



Summary

- GraphChi
  - Optimized for sequential accesses
- FlashGraph
  - Vertex data in memory to handle random access
- Mosaic
  - Xeon Phi to parallelize I/O and computation
- BigSparse
  - Sort-Reduce to remove random access