X-Stream

Edge-centric Graph Processing using Streaming Partitions

Roy et al., EPFL, 2018

Jan Pytel
University of Cambridge
November 5, 2025

X-Stream: Introduction & Motivation

- Graph processing system for:
  - in-memory graphs
  - out-of-core graphs
- Single, shared-memory machine
- Implements Scatter-Gather programming model
- Novelty
  - **edge-centric** (vertex-centric approach most common)
  - **streaming unordered edge list**
    - from RAM (in-memory graphs)
    - from disk (out-of-core graphs)
- Motivation: **sequential access** > random access bandwidth

Scatter-Gather

- Graph (vertices & edges)
  - state stored in vertices
- Iterative computation:
  1. scatter vertex state to neighbours
  2. gather updates from neighbours and recompute vertex state
- Vertex-Centric implementation
  - iterates over vertices
  - issue: random access for edges
- Edge-Centric implementation
  - iterates over edges
  - benefit: sequential access to edge list

Random vs Sequential Access

- Read bandwidth comparison (sequential vs random)
  - disk: 500x faster
  - SSD: 30x faster
  - RAM:
    - 4.6x faster (single core)
    - 1.8x faster (16 cores)
    - due to **hardware prefetching**
- X-Stream: exploit sequential access

Streams

- Scatter phase
  - takes: stream of edges
  - produces: stream of updates
- X-Stream: stream edge list from *slow storage*
- In-memory graphs
  - *Fast Storage*: cache
  - *Slow Storage*: RAM
- Out-of-core graphs
  - *Fast Storage*: RAM
  - *Slow Storage*: SSD/disk

Streams

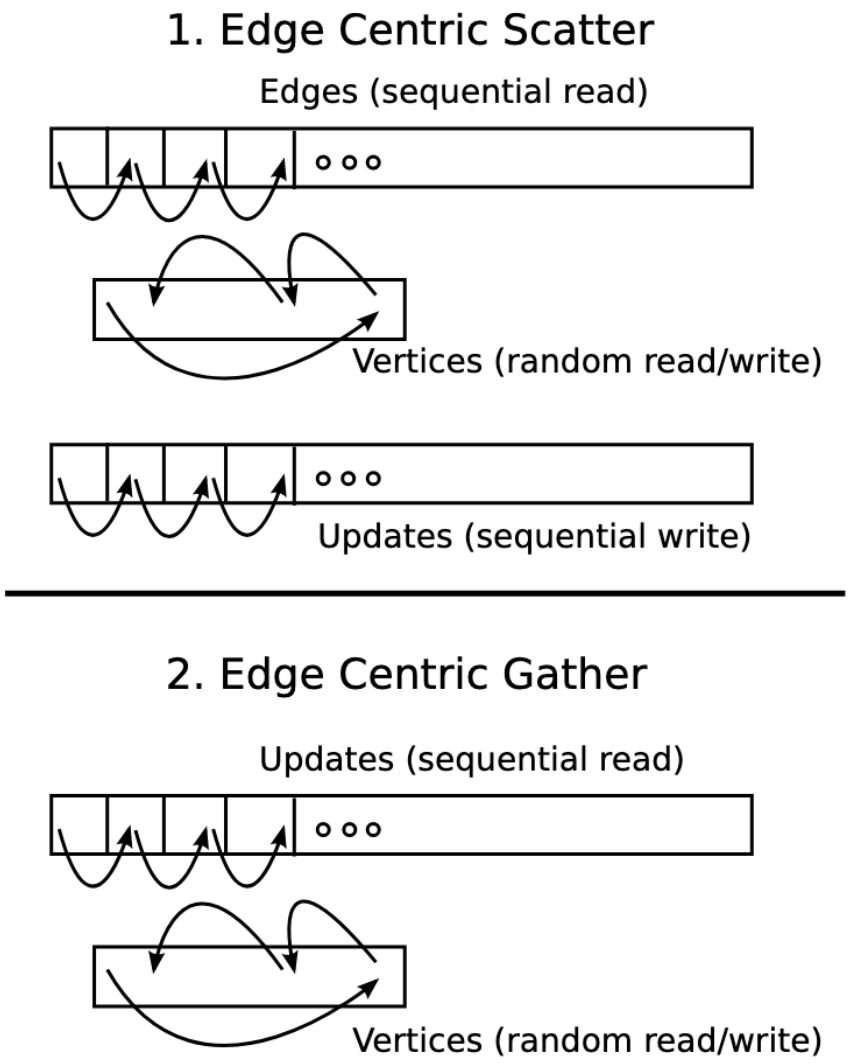

Figure 3: Streaming Memory Access

Streaming Partitions

- Goal: avoid random access to vertices
- Typical graph: |edge set| > |vertex set|
  - still want to avoid random access to vertices
- Large graphs: vertex set may not fit into *Fast Storage*
- Solution: **Streaming Partitions**
  - idea: split vertex set into subsets that fit into *Fast Storage*
  - consists of
    - **vertex set** – subset of vertices
    - **edge list** – edges whose source vertex is in the partition's vertex set
    - **update list** – updates whose destination vertex is in partition's vertex set
      - recomputed before each gather phase

Scatter-Gather w/ Streaming Partitions

- Scatter & Gather phases
  - iterate over streaming partitions (not edges)
- Shuffle phase (new)
  - recompute update list

```
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)
```

Figure 4: Edge-Centric Scatter-Gather with Streaming Partitions

Size and Number of Partitions

- Goal
  - fast random access to vertices: fit all vertices into *Fast Memory*
  - maximise sequential access to *Slow Storage*: minimal num of partitions
- Solution
  - vertex sets of equal size
  - vertex sets fills up *Fast Storage*
    - (allowing for buffers and additional data structures)

Out-of-core Streaming Engine

- Input: file with unordered edge list
- 3 files for each streaming partition
  - 1 for vertices
  - 1 for edges
  - 1 for updates
- Challenge: achieve sequential access for shuffle phase
  - random I/O if tried to write updates to files as they come
  - solution: “fold” shuffle phase into scatter phase
    - **in-memory buffer** storing updates
    - run **in-memory shuffle** when full

Stream Buffer

- Statically sized, statically allocated
- 5 stream buffers:
  - 2 input and 2 output buffers (to support prefetching)
  - 1 for shuffling

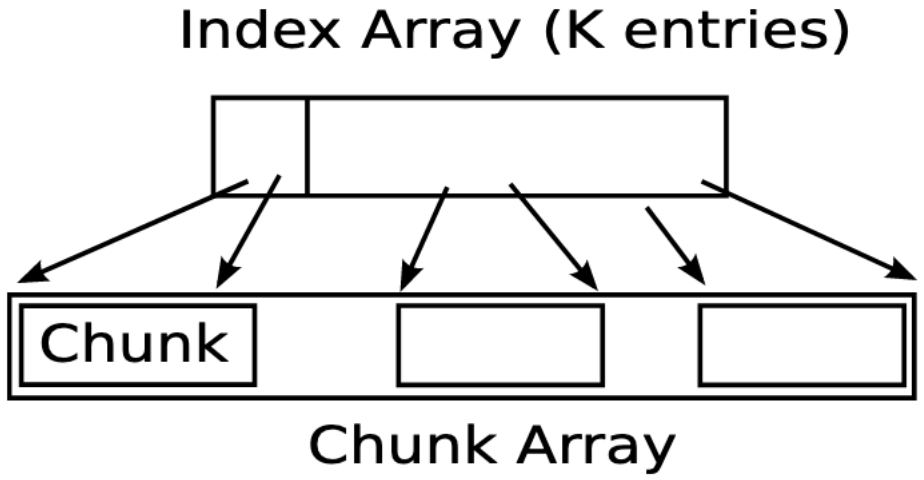

Figure 5: Stream Buffer (K: number of partitions)

Out-of-core Streaming Engine: Operation

- Pre-process: partition input edge lists into streaming partitions
- Main loop

```
merged scatter/shuffle phase:
  for each streaming partition s
    while edges left in s
      load next chunk of edges into input buffer
      for each edge e in memory
        edge_scatter(e) appending to output buffer
        if output buffer is full or no more edges
          in-memory shuffle output buffer
          for each streaming partition p
            append chunk p to update file for p

gather phase:
  for each streaming_partition p
    read in vertex set of p
    while updates left in p
      load next chunk of updates into input buffer
      for each update u in input buffer
        edge_gather(u)
    write vertex set of p
```

Figure 6: Disk Streaming Loop

In-memory Streaming Engine

- Concerns
  - parallelism: need all cores for peak streaming bandwidth to memory
  - larger number of partitions (aim: fit vertex set in CPU cache)
- 3 stream buffers
  - Input (edges), output (updates), shuffling
- Operation
  - load edges into input stream buffer
  - shuffle edges into chunk of edges for each partition
  - append all updates to a single output buffer
  - shuffle updates into chunks of updates for each partition

In-memory Streaming Engine: Parallelism

- Parallel Scatter-Gather
  - streaming operations done independently for separate partitions
  - challenge: workload imbalance if num of edges differs per partition
  - solution: “work stealing” (threads ”stealing” partitions from each other)
- Parallel Multistage Shuffler
  - issue: shuffling into large number of partitions inefficient
    - (loss of sequential access)
  - idea: group partitions into tree hierarchy and shuffle more efficiently
    - shuffles K partitions in $\lceil \log_F K \rceil$ steps
  - multithreading – stream buffer slices, thread per slice

In-memory Streaming Engine: Parallelism

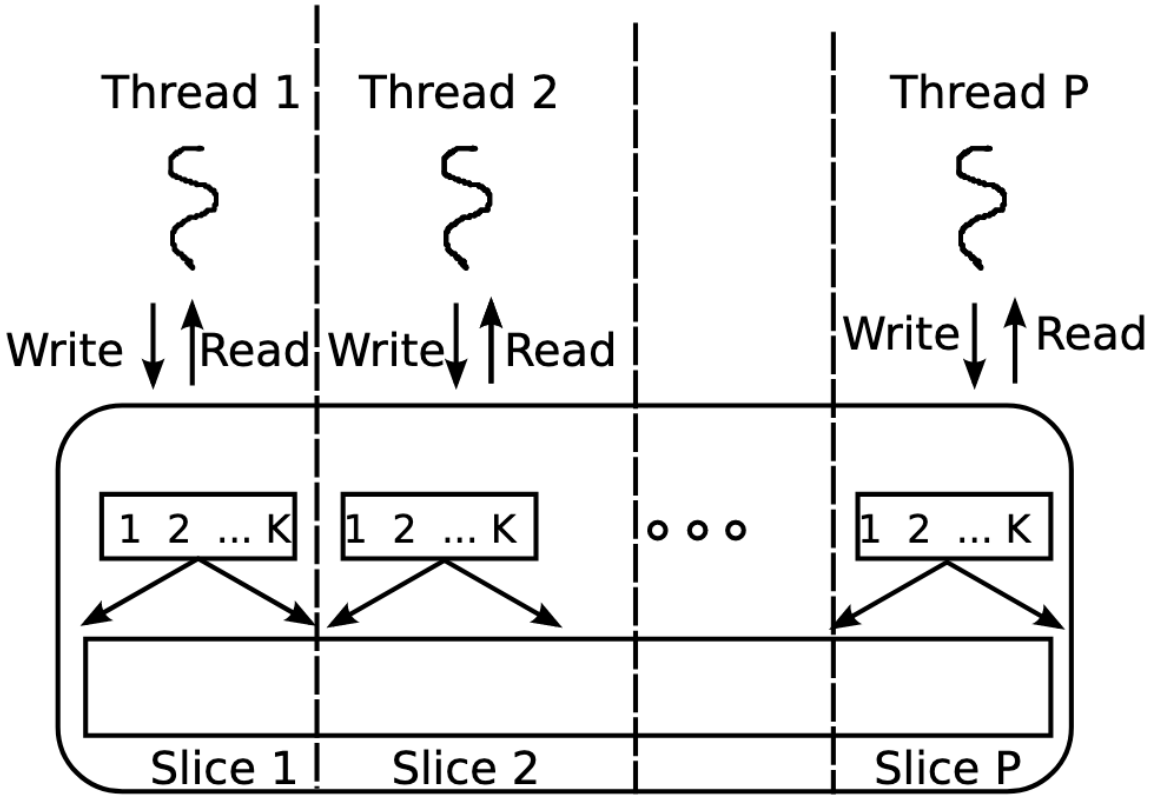

Figure 7: Slicing a Streaming Buffer

X-Stream Layering

- In-memory engine above out-of-core engine
- For each iteration of the disk streaming loop
  - loaded input chunk processed by in-memory engine
- Maximises main memory bandwidth usage and computational resources with the out-of-core engine

Evaluation

Name	Vertices	Edges	Type
	In-memory		
amazon0601 [2]	403,394	3,387,388	Directed
cit-Patents [3]	3,774,768	16,518,948	Directed
soc-livejournal [4]	4,847,571	68,993,773	Directed
dimacs-usa [5]	23,947,347	58,333,344	Directed
	Out-of-core		
Twitter [36]	41.7 million	1.4 billion	Directed
Friendster [6]	65.6 million	1.8 billion	Undir.
sk-2005 [7]	50.6 million	1.9 billion	Directed
yahoo-web [8]	1.4 billion	6.6 billion	Directed
Netflix [55]	0.5 million	0.1 billion	Bipartite

Figure 10: Datasets

Evaluation

	WCC	SCC	SSSP	MCST	MIS	Cond.	SpMV	Pagerank	BP
memory									
amazon0601	0.61s	1.12s	0.83s	0.37s	3.31s	0.07s	0.09s	0.25s	1.38s
cit-Patents	2.98s	0.69s	0.29s	2.35s	3.72s	0.19s	0.19s	0.74s	6.32s
soc-livejournal	7.22s	11.12s	9.60s	7.66s	15.54s	0.78s	0.74s	2.90s	1m 21s
dimacs-usa	**6m 12s**	**9m 54s**	**38m 32s**	4.68s	9.60s	0.26s	0.65s	2.58s	12.01s
ssd									
Friendster	38m 38s	1h 8m 12s	1h 57m 52s	19m 13s	1h 16m 29s	2m 3s	3m 41s	15m 31s	52m 24s
sk-2005	44m 3s	1h 56m 58s	2h 13m 5s	19m 30s	3h 21m 18s	2m 14s	1m 59s	8m 9s	56m 29s
Twitter	19m 19s	35m 23s	32m 25s	10m 17s	47m 43s	1m 40s	1m 29s	6m 12s	42m 52s
disk									
Friendster	1h 17m 18s	2h 29m 39s	3h 53m 44s	43m 19s	2h 39m 16s	4m 25s	7m 42s	32m 16s	1h 57m 36s
sk-2005	1h 30m 3s	4h 40m 49s	4h 41m 26s	39m 12s	7h 1m 21s	4m 45s	4m 12s	17m 22s	2h 24m 28s
Twitter	39m 47s	1h 39m 9s	1h 10m 12s	29m 8s	1h 42m 14s	3m 38s	3m 13s	13m 21s	2h 8m 13s
yahoo-web	—	—	—	—	—	16m 32s	14m 40s	1h 21m 14s	8h 2m 58s

(a)

	# iters	ratio	wasted %
memory			
amazon0601	19	2.58	63
cit-Patents	21	2.20	50
soc-livejournal	13	2.13	57
dimacs-usa	**6263**	1.94	**98**
ssd			
Friendster	24	1.06	63
sk-2005	25	1.04	67
Twitter	16	1.04	55
disk			
Friendster	24	1.04	63
sk-2005	25	1.04	67
Twitter	16	1.04	55
yahoo-web	—	—	—

(b)

Figure 12: Different Algorithms on Real World Graphs: (a) Runtimes; (b) Number of scatter-gather iterations, ratio of runtime to streaming time, and percentage of wasted edges for WCC.

Scalability

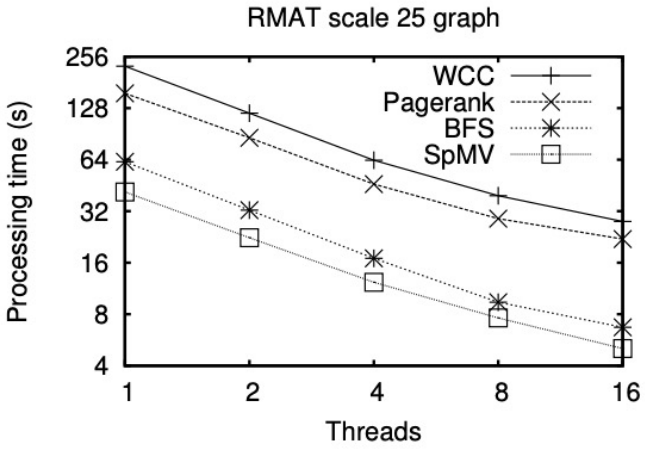

Figure 14: Strong Scaling

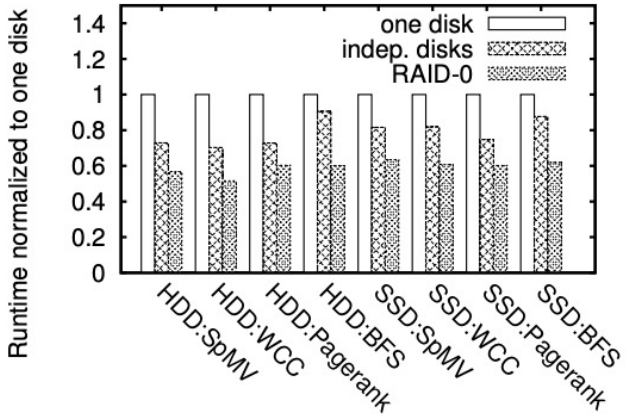

Figure 15: I/O Parallelism

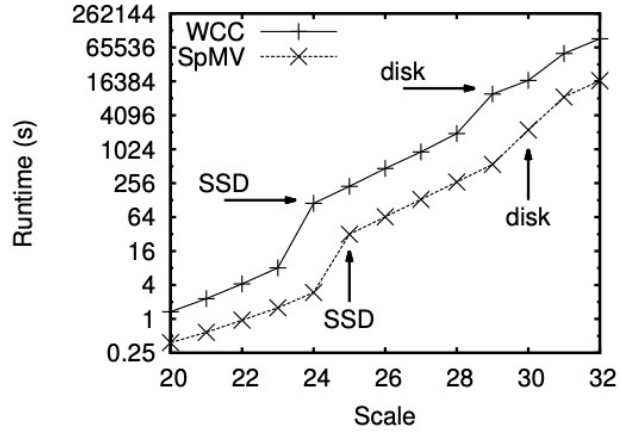

Figure 16: Scaling Across Devices

X-Stream vs GraphChi (Out-of-Core)

	X-Stream	GraphChi
Model	edge-centric	vertex-centric
Key Idea	eliminate random I/O by streaming edges and updates	reduce random I/O by sorting edges into shards per vertex
I/O Strategy	sequential access streaming of unsorted edge list	“parallel sliding windows” to reduce amount of random access to disk
Pre-Processing	none	pre-sorting graph into shards
Access Pattern	fully sequential	semi-sequential (multiple shards accessed per iteration)
Primary Limitation	limited random access to vertices	pre-processing can dominate runtime
Scalability	scales linearly with graph size, I/O-bound, not CPU-bound	works well on large graphs, however costly pre-processing

X-Stream vs GraphChi

	Pre-Sort (s)	Runtime (s)	Re-sort (s)
Twitter pagerank			
X-Stream (1)	*none*	397.57 ± 1.83	–
Graphchi (32)	752.32 ± 9.07	1175.12 ± 25.62	969.99
Netflix ALS			
X-Stream (1)	*none*	76.74 ± 0.16	–
Graphchi (14)	123.73 ± 4.06	138.68 ± 26.13	45.02
RMAT27 WCC			
X-Stream (1)	*none*	867.59 ± 2.35	–
Graphchi (24)	2149.38 ± 41.35	2823.99 ± 704.99	1727.01
Twitter belief prop.			
X-Stream (1)	*none*	2665.64 ± 6.90	–
Graphchi (17)	742.42 ± 13.50	4589.52 ± 322.28	1717.50

Figure 22: Comparison with Graphchi on SSD with 99% Confidence Intervals. Numbers in brackets indicate X-Stream streaming partitions/Graphchi shards (Note: re-sorting is included in Graphchi runtime.)

Summary

- Outperforms existing graph processing systems
- Key performance factors
  - sequential access
  - no pre-processing cost (e.g. sorting & indexing)
  - higher count of instruction per cycle (lower memory resolution latency)
- Scalability – scales well across
  - number of cores; number of I/O devices; different storage devices
- Discussion
  - I/O centric design philosophy
  - Simplicity/semantic tradeoff
  - Pre-processing tradeoff
  - High-diameter graphs and “wasted edges”
  - Single-machine assumption