X-Stream: Edge-centric Graph Processing using Streaming Partitions

Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel

Motivation

- Graph processing is widely used
  - Especially on huge graphs (billions of vertices, trillions of edges)
- Existing systems such as Pregel and PowerGraph have a good programming model
  - Implementation results in many random memory accesses
  - Sequential accesses to edges could result in better performance

Scatter-Gather Programming Model

- Maintain state in vertices
- *Scatter* - propagate vertex state to neighbours
- *Gather* - cumulate updates and recompute state
- Approach is **vertex-centric**

```
vertex_scatter(vertex v)
      send updates over outgoing edges of v

vertex_gather(vertex v)
      apply updates from inbound edges of v

while not done
      for all vertices v that need to scatter updates
            vertex_scatter(v)
      for all vertices v that have updates
            vertex_gather(v)
```

Vertex-centric vs Edge-centric

Vertex-centric	**Edge-centric**
Iterates over vertices	Iterates over edges
Random accesses of edges	Sequential accesses of edges
Requires pre-sorting to generate indexed edge list	Requires no pre-sorting
Edges only used when needed	All edges used per iteration

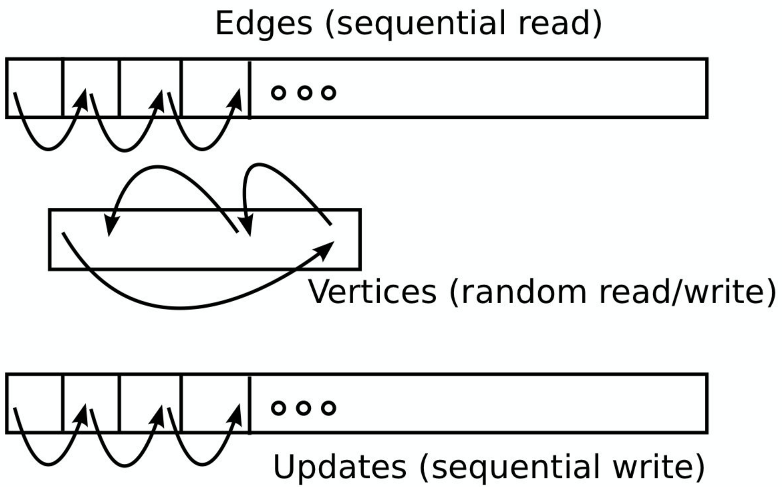
Edge-centric Model

- Approach is **edge-centric**
- Still maintain state in vertices
- Can stream edges and updates from storage - *no longer random access to edges*
- We now have random access to vertices

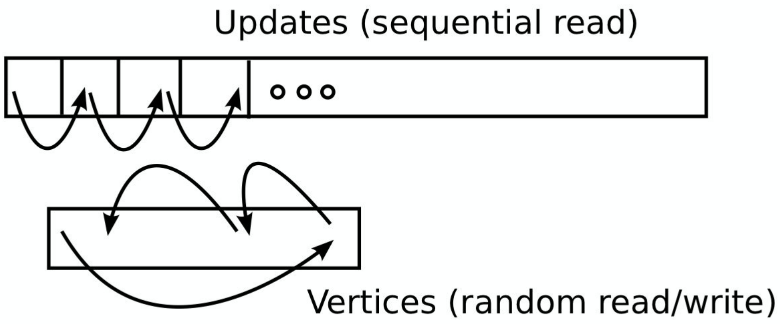
```
edge_scatter(edge e)
      send update over e

edge_gather(update u)
      apply update u to u.destination

while not done
      for all edges e
            edge_scatter(e)
      for all updates u
            update_gather(u)
```

Edge–centric Model

1. Edge Centric Scatter



2. Edge Centric Gather



Streaming Partitions

- Comprised of 3 elements
  - *Vertex set* - a subset of vertices from the graph
  - *Edge list* - all edges where the *source* vertex is in the partition’s vertex set
  - *Update list* - all updates where the *destination* vertex is in the partition’s vertex set
- Vertex set exists in a *cache*
  - Mitigates the random access issue
  - Size must be balanced though

Streaming Partitions

- Must now introduce a new shuffle phase
    - Destination of update may not reside in the same streaming partition
    - Shuffle Phase reorders the updates so this is the case
    - Have `Uout` and `Uin` stream

```
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)
```

Two scenarios for X-Stream

In-memory

- *Fast Storage* = CPU cache
- *Slow Storage* = Main Memory
- Very limited by size of storage

Out-of-core

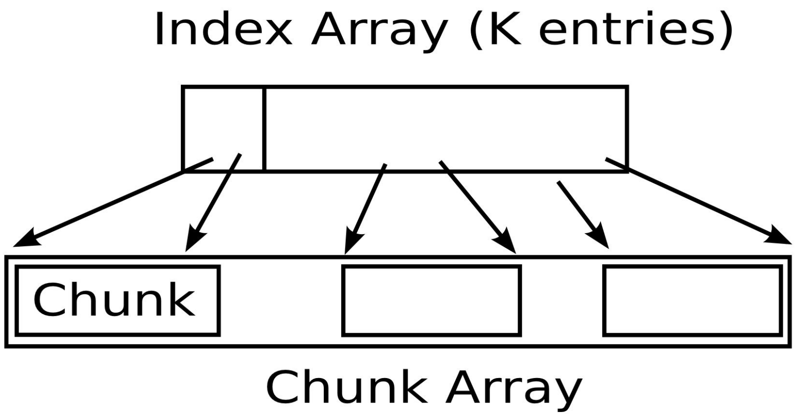
- *Fast Storage* = Main Memory
- *Slow Storage* = SSD or Hard Disk
- Huge capacity of storage

In-Memory Specifics

- Cache is much smaller than main memory
  - Results in more streaming partitions needed
- Need to go parallel to reach peak streaming bandwidth
  - Each partition can operate independently
    - Can result in workload imbalance
    - Fix by stealing jobs

Out-of-Core Specifics

- Merge scatter and shuffle phase
  - Only shuffle when output buffer is full
- Use a stream buffer to store updates
  - Makes shuffling more efficient - *O(n)*
  - Requires two stream buffers
    - Input to shuffle
    - Output of shuffle
  - Also used for partitioning



Evaluation

- Sequential Access is better than Random Access on every medium

Medium	Read (MB/s)		Write (MB/s)	
	Random	Sequential	Random	Sequential
RAM (1 core)	567	2605	1057	2248
RAM (16 cores)	14198	25658	10044	13384
SSD	22.5	667.69	48.6	576.5
Magnetic Disk	0.6	328	2	316.3

Figure 11: Sequential Access vs. Random Access

Evaluation

- Initial results were poor for some datasets
    - The *DIMACS* and *Yahoo web-graph* datasets have a wide diameter
    - Results in more scatter-gather iterations as information passes from one end to the other
    - Each iteration requires entire edge list to be streamed; not much useful work each time

Evaluation

- Demonstrates strong scaling with regard to number of threads and I/O Parallelism

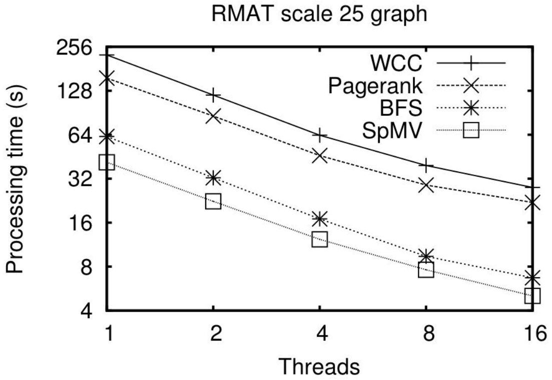


Figure 14: Strong Scaling

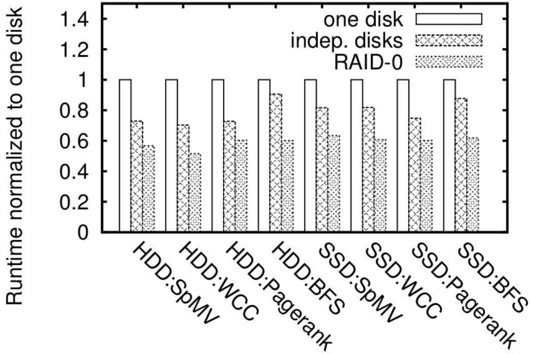


Figure 15: I/O Parallelism

Evaluation – In Memory

- Compared to Ligra
  - BFS and Pagerank were used to test

- For BFS without preprocessing times Ligra performs better
  - X-Stream is much faster when pre-processing times taken into account

- Due to sequential access, IPC was much higher

Threads	Ligra (s)	X-Stream (s)	Ligra-pre (s)
	BFS		
1	11.10	168.50	1250.00
2	5.59	86.97	647.00
4	2.83	45.12	352.00
8	1.48	26.68	209.40
16	0.85	18.48	157.20
	Pagerank		
1	990.20	455.06	1264.00
2	510.60	241.56	654.00
4	269.60	129.72	355.00
8	145.40	83.42	211.40
16	79.24	50.06	160.20

Figure 20: Ligra [48] on Twitter (99%CI under 5%)

	BFS [33]	X-Stream
IPC Mem refs.	0.47 982 million	1.30 620 million
	Ligra,BFS [48]	X-Stream
IPC Mem refs.	0.75 1.3 billion	1.39 1.5 billion

Figure 21: Instructions per Cycle and Total Number of Memory References for BFS

Evaluation - Out-of-Core

- Compared to GraphChi
- GraphChi needs to resort edges by destination vertex before applying updates (reported as `re-sort`)
    - Accounts for significant proportion of execution
- In most cases X-Stream completed before GraphChi Presorted
- Disk bandwidth of X-Stream is much greater

	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.)

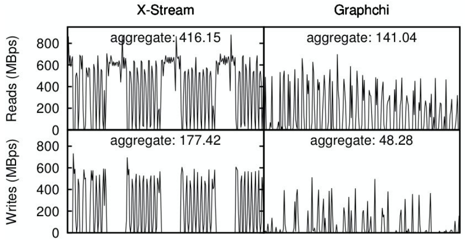


Figure 23: Disk Bandwidth

Strengths and Weaknesses

Strengths

- Utilises sequential access to effectively maximise streaming bandwidth
- Performs quicker in general than most state of the art systems on some algorithms

Weaknesses

- If no pre-processing is required for the vertex-centric approach, performance can be weaker
- Performance is poor on a graph with a high diameter
- “Wasted Edges” also negatively impacts performance

Criticism

- Little focus on running X-Stream on multiple machines
  - *See next slide*

- Assumes graph has many more edges than vertices

- No discussion on the limitations of the programming model

Impact

- Nearly 450 citations
- Authors extended X-Stream with *Chaos*
    - *Chaos essentially is the distributed version of X-Stream*

Questions