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

Amitabha Roy, Ivo Mihailovic,
Willy Zwaenepoel

Context

Approach

Model

Implementation

Results & Conclusion

Pregel & Powergraph: *scatter & gather*

→ A *scatter-gather* methodology:

1. scatter(vertex v):

   send updates over outgoing edges of v

2. gather(vertex v):

   apply updates from inbound edges of v

→ *how to* scale-up?

Trade-off: *Sequential vs Random access*

Medium	Read (MB/s) Random	Read (MB/s) Sequential	Write (MB/s) Random	Write (MB/s) 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

GraphChi: a sequential approach

→ avoids random access using *shards*

Problems:

1. need graph to be pre-sorted by source vertex
2. vertex-centric
3. requires re-sort of edges by destination vertex for gather step

Context

Approach

Model

Implementation

Results & Conclusion

X-Stream's Approach

1. retain *scatter-gather* programming model
2. use an edge-centric implementation
3. stream unordered edge lists

Gains:

1. use sequential (*not* random) access
2. do not need pre-processing step

scatter-gather: an edge-centric implementation

```
scatter(edge e):
    send update over e

gather(update u):
    apply update u to u.destination
```

Quick Terminology

Fast Storage:

→ caches (in-memory)

→ main-memory (out-of-core)

Slow Storage:

→ main-memory (in-memory)

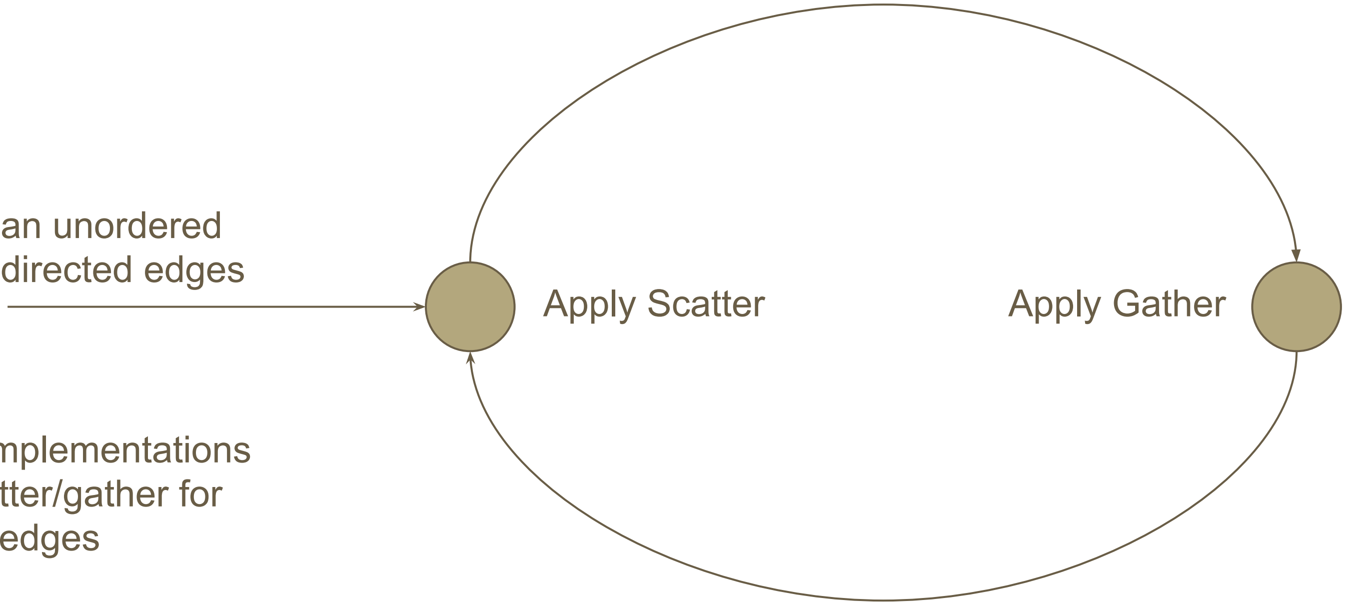
→ SSD/Disk (out-of-core)

Context
Approach
Model
Implementation
Results & Conclusion

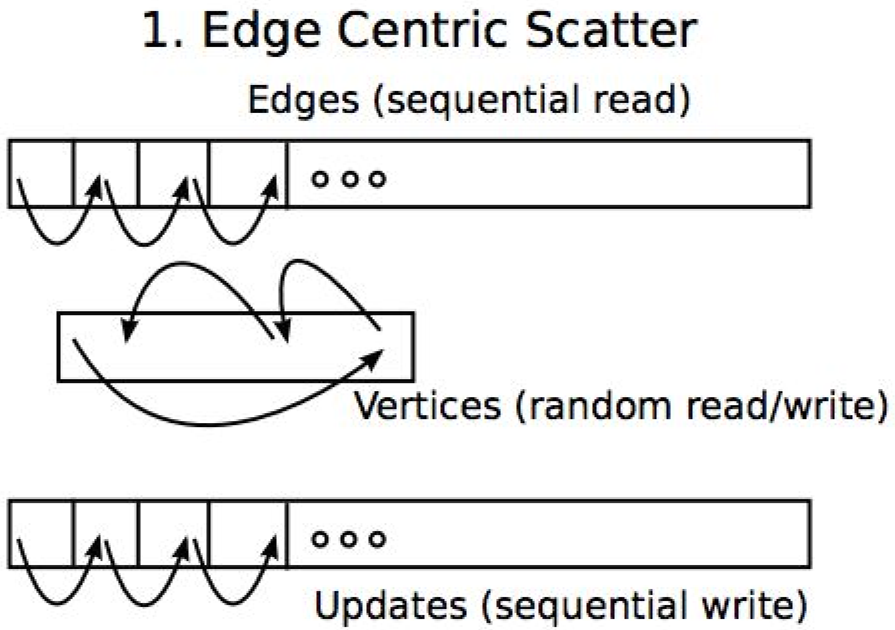
The basic model:

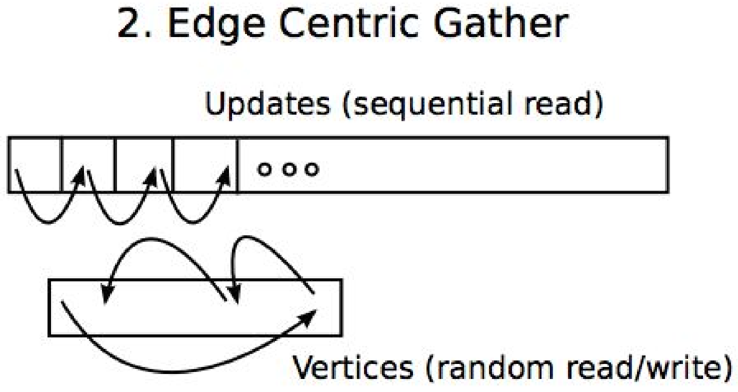
input: an unordered set of directed edges

API: implementations of scatter/gather for given edges



Problem: vertices may *not* fit in fast storage





Problem: vertices may *not* fit in fast storage

→ Streaming partitions:

- vertex set, V: a subset of the vertices of the graph
- edge list: source is $\in$ V
- update list: dest $\in$ V

→ How do we use them?

1. scatter/gather iterate over streaming partitions
2. updates need to be *shuffled*

Context

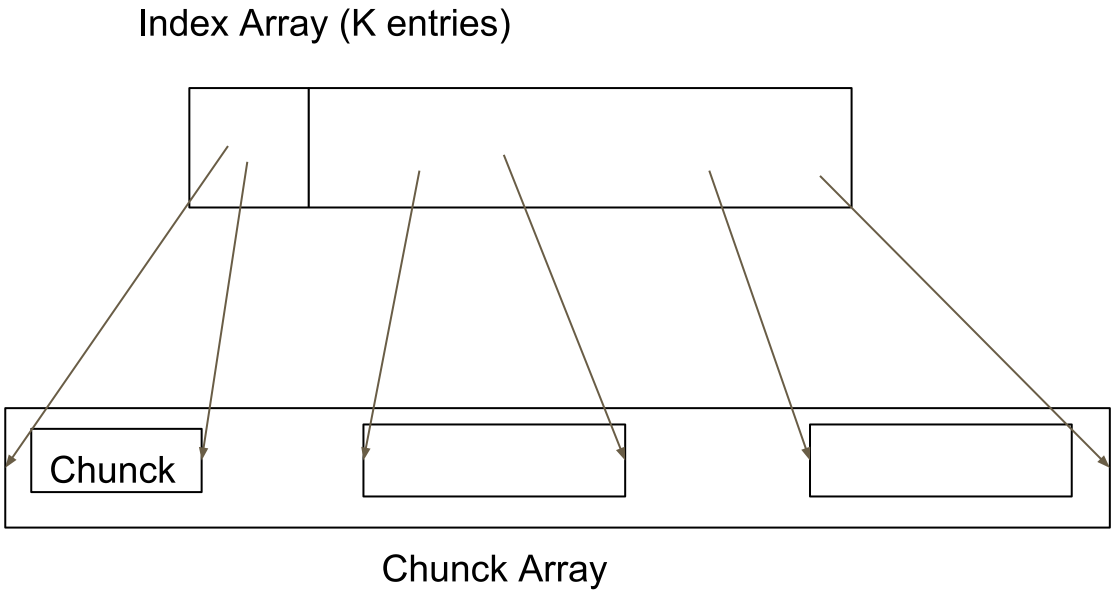
Approach

Model

Implementation

Results & Conclusion

Stream buffer



Out-of-core

→ Folds shuffle into scatter

- run scatter, appending updates to an in-memory buffer
- when buffer full: run an *in-memory shuffle*

→ 2 Stream Buffers

→ Number of partitions

N/K + 5SK <= M

→ Disk I/O

In-memory

→ Parallel multi-stage shuffler & scatter/gather

- stream independently for each streaming partition
- work stealing
- group partitions together into a tree for the shuffler

→ 3 stream buffers

→ Number of partitions

= CPU_cache_size / footprint

Chaos: the extension of X-Stream

→ Scale out to multiple machines in 1 cluster

2 gains:

1. access secondary storage in parallel improves performance
2. increases size of graph that can be handled

Chaos: the extension of X-Stream

→ Steps:

1. simple initial partitioning
2. spread graph data uniformly over all 2nd storage devices
3. work stealing

Assumptions:

1. network machine-to-machine bandwidth > bandwidth of storage device
2. network switch bandwidth > aggregate bandwidth of all storage devices of cluster

Context

Approach

Model

Implementation

Results & Conclusion

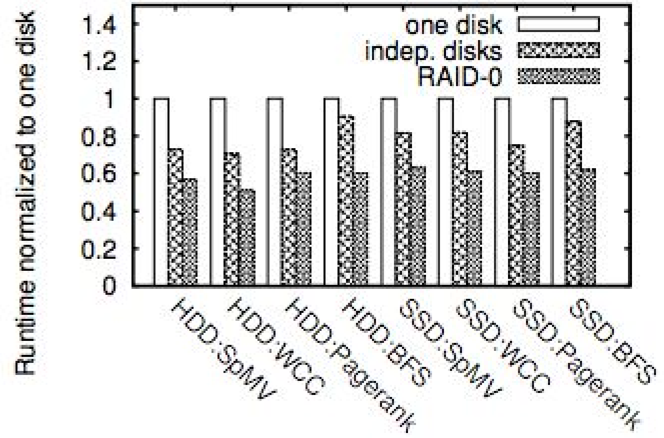
Experiments:

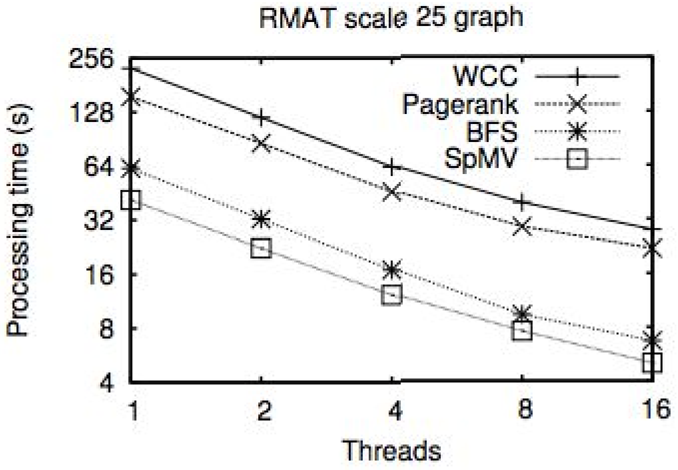
→ Tested on real-world graphs.

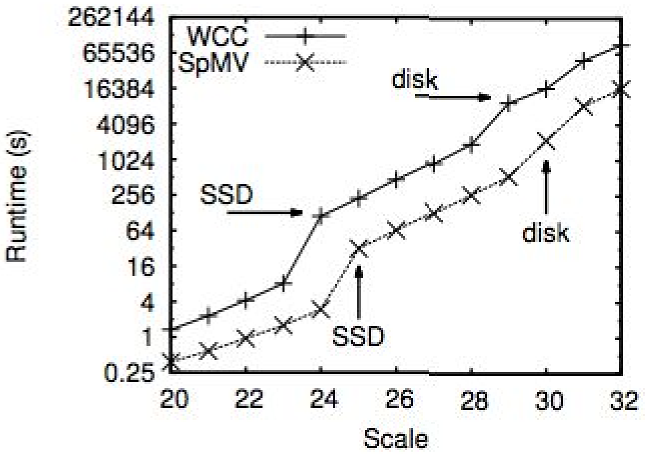
	WCC	SCC	SSSP	MCST	MIS	Cond.	SpMV	Pagerank	BP
memory									
amazon0601	0.61s	1.12s	0.83s	0.37s	3.31s	0.07s	0.09s	0.25s	1.38s
cit-Patents	2.98s	0.69s	0.29s	2.35s	3.72s	0.19s	0.19s	0.74s	6.32s
soc-livejournal	7.22s	11.12s	9.60s	7.66s	15.54s	0.78s	0.74s	2.90s	1m 21s
dimacs-usa	**6m 12s**	**9m 54s**	**38m 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)

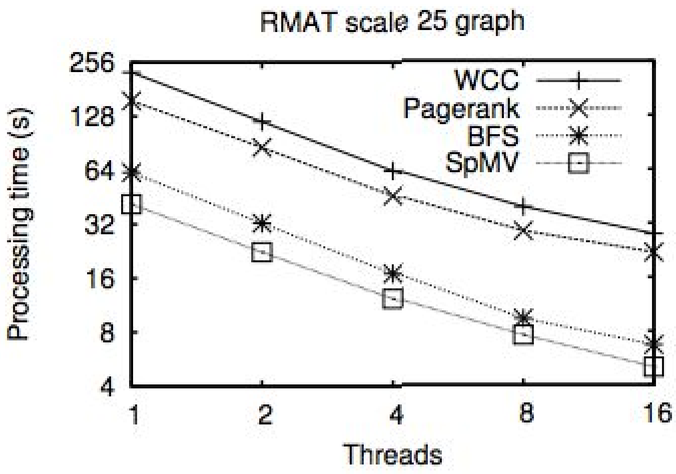
Scalability



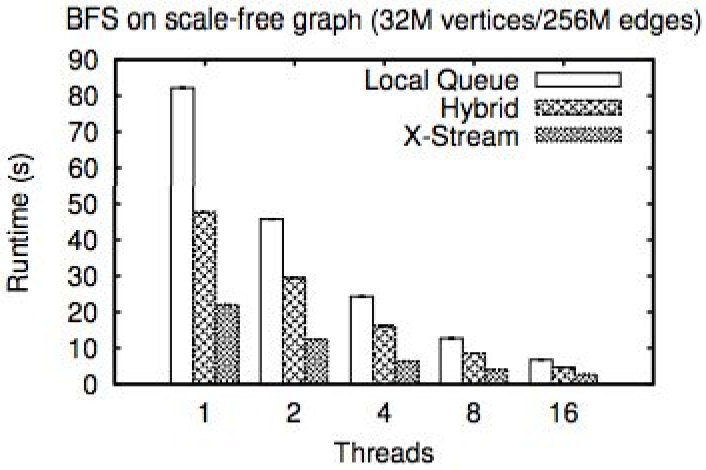




Comparison



RMAT scale 25 graph
Processing time (s)
256
128
64
32
16
8
4
1
2
4
8
16
Threads
WCC
Pagerank
BFS
SpMV



BFS on scale-free graph (32M vertices/256M edges)
Runtime (s)
90
80
70
60
50
40
30
20
10
0
1
2
4
8
16
Threads
Local Queue
Hybrid
X-Stream

Comparison: Ligra

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

Comparison: 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

Conclusion & Takeaway

Strengths:

→ Sequential access

→ Scale up & scale out

Weaknesses

→ Limited number of problems it can handle

→ Limited types of graphs it can handle

→ How would you use in a real-world scenario