GraphChi: Large-Scale Graph Computation on Just a PC

By Aapo Kyrola, Guy Blelloch, & Carlos Guestrin
Presented by Lucas Bautista

Aapo Kyrola

Guy Blelloch

Carlos Guestrin

Processing LARGE graphs

- Main problem is that large graphs don’t fit into main memory
  - Distributed Systems?

- Many problems with distributed systems
  - Complexity
  - Optimal Load Balancing (partitioning)
  - Fault Tolerance

Processing LARGE graphs

- Main problem is that large graphs don’t fit into main memory
  - Distributed Systems?

- Many problems with distributed systems
  - Complexity
  - Optimal Load Balancing (partitioning)
  - Fault Tolerance

- Use Pregel
  - Synchronization is expensive

- Use PowerGraph/GraphLab
  - …

Realistically though, some large graphs can fit in most computer disks + network communication might be slower(?)

Computational Model

We use a vertex-centric model of computation (like in Pregel)

G = (V, E) **Directed + Sparse**

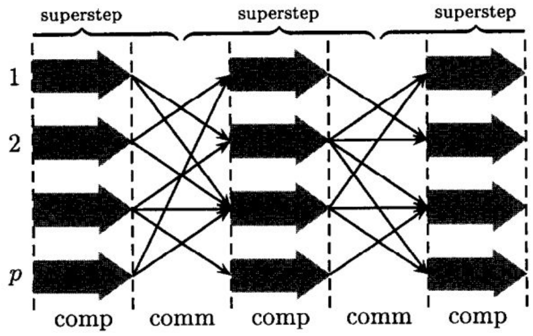
Each vertex & edge has an associated value

- Vertices numbered from 1 to |V|
- User specified **update-function**
  - Can modify edge & vertex values
- Executed until a termination condition is satisfied

No explicit message sending.

Bulk Synchronous Parallel

- Parallel executions occur in lockstep (superstep idea / global barrier).
- Once all parallel executions terminate in a superstep, we can start a new superstep with the new generated values.



Downside: The synchronization step at the end of a superstep is costly

We trade off some parallelism for an asynchronous model.

Asynchronous model

- Each vertex can use the most recent values of edges
- Uses dynamic selective scheduling to only process nodes that will change because of updated values (similar to idle vs active nodes in the Pregel model).

Asynchronous model

- Each vertex can use the most recent values of edges
- Uses dynamic selective scheduling to only process nodes that will change because of updated values (similar to idle vs active nodes in the Pregel model).

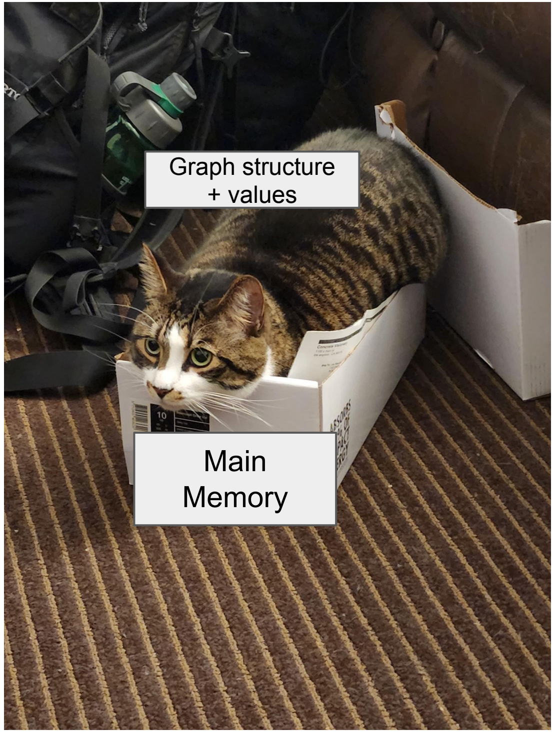
Problem: Concurrent reads & writes

- Sequential execution for vertices that share an edge

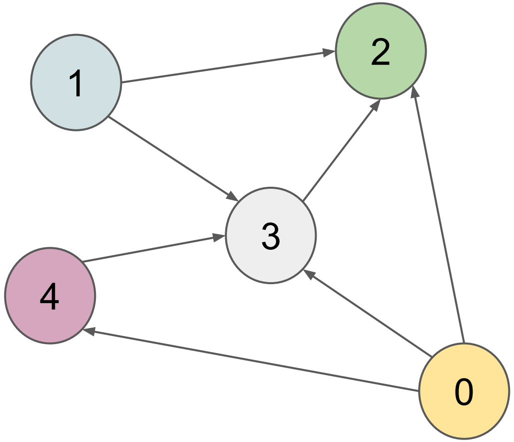
GraphChi

Assumptions

- Graph structure, vertex values, and edge values do not fit in main memory
- There is enough memory to contain the edges and their associated values of a **single** vertex
- All active windows of shards fit in main memory



CSR



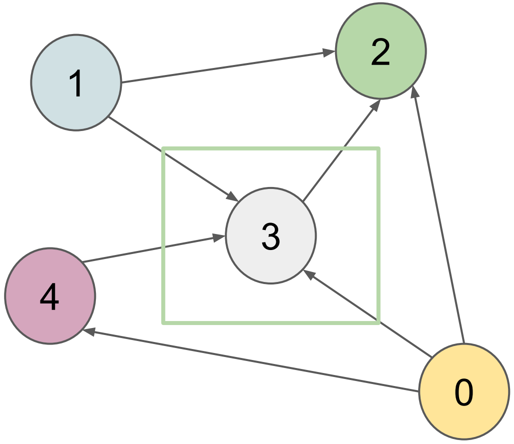
Row Array

0	3	5	5	6	7
0	1	2	3	4	5

Column Array

2	3	4	2	3	2	3
0	1	2	3	4	5	6

CSR



Row Array

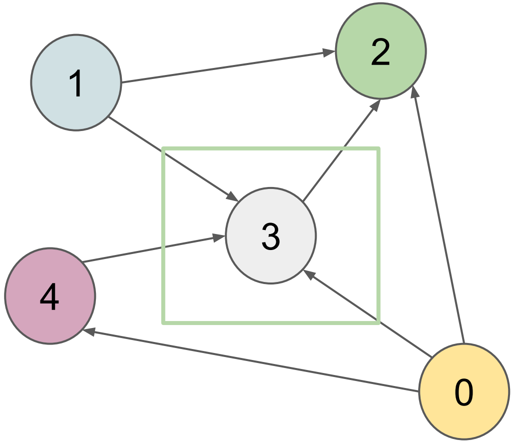
0	3	5	5	6	7
0	1	2	3	4	5

Column Array

2	3	4	2	3	2	3
0	1	2	3	4	5	6

But we need to access the in-edges of a vertex as well, which is not easy to get from CSR form.

CSR



Row Array

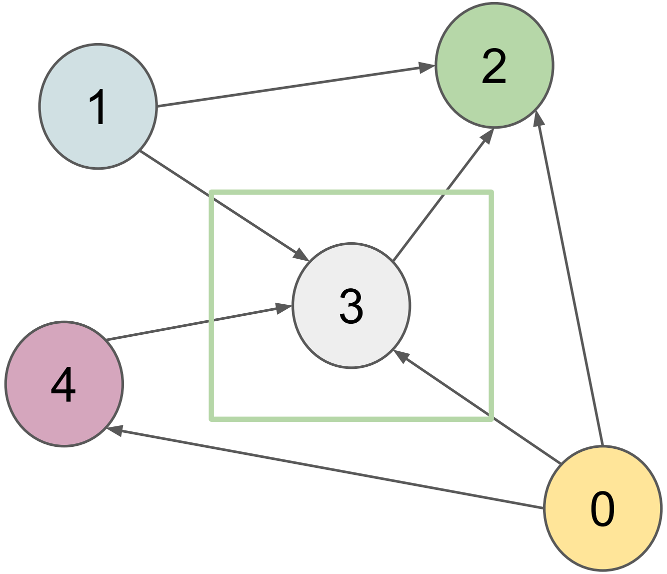
0	3	5	5	6	7
0	1	2	3	4	5

Column Array

2	3	4	2	3	2	3
0	1	2	3	4	5	6

But we need to access the in-edges of a vertex as well, which is not easy to get from CSR form.

CSR + CSC



CSC (in-edges)

Row Array

0	0	0	3	6	7
0	1	2	3	4	5

Column Array

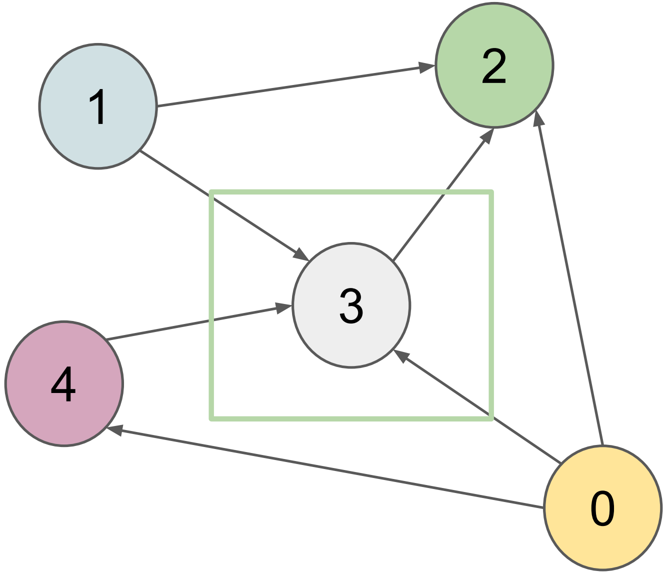
0	1	3	0	1	4	0
0	1	2	3	4	5	6

CSR (out-edges)

0	3	5	5	6	7
0	1	2	3	4	5

2	3	4	2	3	2	3
0	1	2	3	4	5	6

CSR + CSC



CSC (in-edges)

Row Array

0	0	0	3	6	7
0	1	2	3	4	5

Column Array

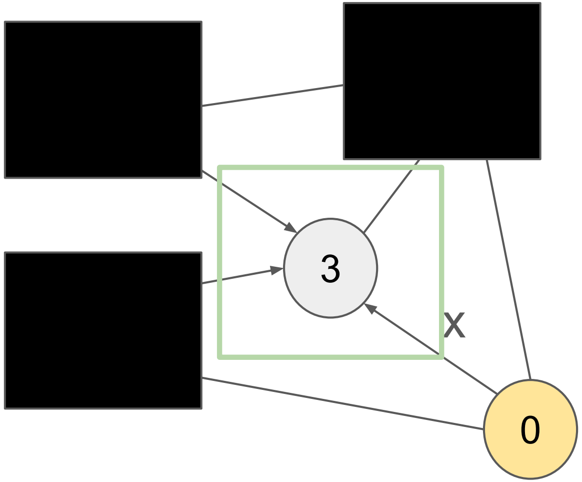
0	1	3	0	1	4	0
0	1	2	3	4	5	6

CSR (out-edges)

0	3	5	5	6	7
0	1	2	3	4	5

2	3	4	2	3	2	3
0	1	2	3	4	5	6

No message passing…



CSC (in-edges)

Row Array

0	0	0	3	6	7
0	1	2	3	4	5

Column Array

0	1	3	0	1	4	0
0	1	2	3	4	5	6

Edge Values

?	?	?	X	?	?	?
0	1	2	3	4	5	6

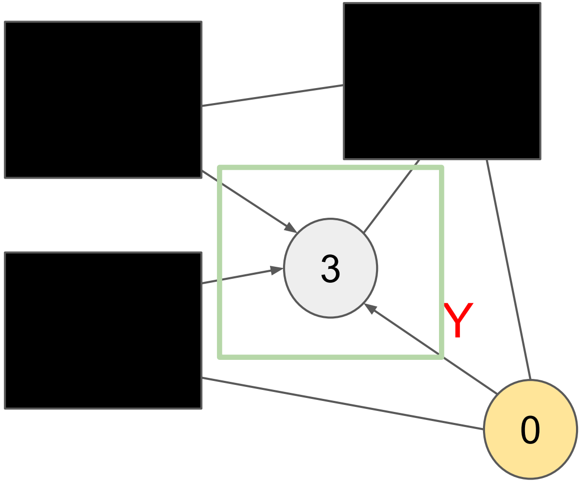
CSR (out-edges)

0	3	5	5	6	7
0	1	2	3	4	5

2	3	4	2	3	2	3
0	1	2	3	4	5	6

?	x	?	?	?	?	?
0	1	2	3	4	5	6

No message passing…



CSC (in-edges)

Row Array

0	0	0	3	6	7
0	1	2	3	4	5

Column Array

0	1	3	0	1	4	0
0	1	2	3	4	5	6

Edge Values

?	?	?	Y	?	?	?
0	1	2	3	4	5	6

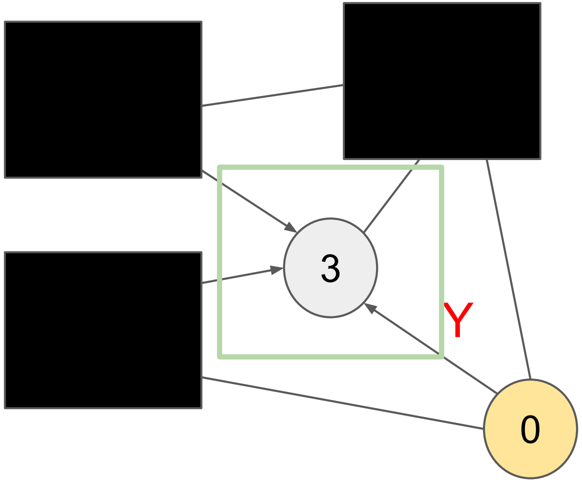
CSR (out-edges)

0	3	5	5	6	7
0	1	2	3	4	5

2	3	4	2	3	2	3
0	1	2	3	4	5	6

?	x	?	?	?	?	?
0	1	2	3	4	5	6

Case 1: **Write to CSR**



CSC (in-edges)

Row Array

0	1	2	3	4	5
0	0	0	3	6	7

Column Array

0	1	2	3	4	5	6
0	1	3	0	1	4	0

Edge Values

0	1	2	3	4	5	6
?	?	?	Y	?	?	?

CSR (out-edges)

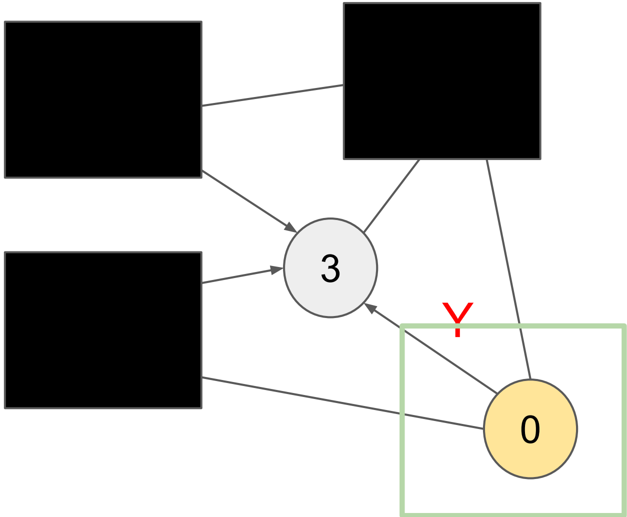
0	1	2	3	4	5
0	3	5	5	6	7

0	1	2	3	4	5	6
2	3	4	2	3	2	3

0	1	2	3	4	5	6
?	Y	?	?	?	?	?

Incurs a random write

Case 1: **Write to CSR**



CSC (in-edges)

Row Array

0	0	0	3	6	7
0	1	2	3	4	5

Column Array

0	1	3	0	1	4	0
0	1	2	3	4	5	6

Edge Values

?	?	?	Y	?	?	?
0	1	2	3	4	5	6

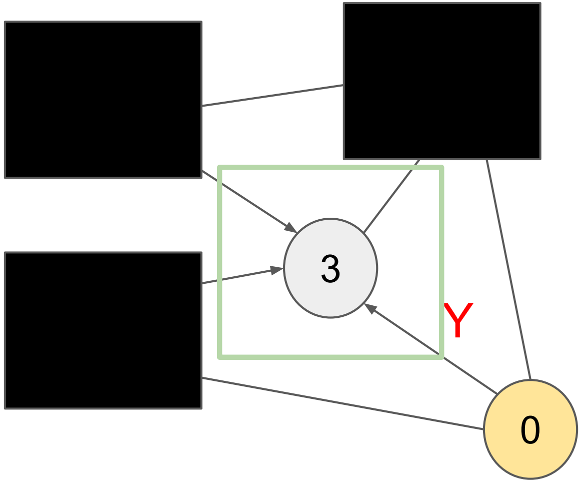
CSR (out-edges)

0	3	5	5	6	7
0	1	2	3	4	5

2	3	4	2	3	2	3
0	1	2	3	4	5	6

?	Y	?	?	?	?	?
0	1	2	3	4	5	6

Case 2: **Read from CSC**



CSC (in-edges)

Row Array

0	1	2	3	4	5
0	0	0	3	6	7

Column Array

0	1	2	3	4	5	6
0	1	3	0	1	4	0

Edge Values

0	1	2	3	4	5	6
?	?	?	Y	?	?	?

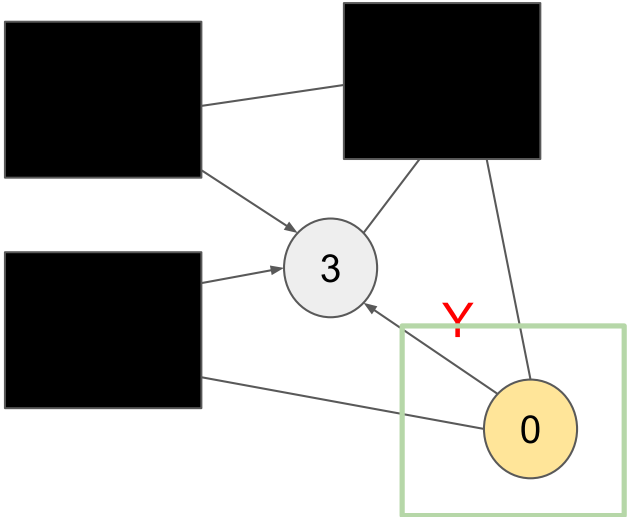
CSR (out-edges)

0	1	2	3	4	5
0	3	5	5	6	7

0	1	2	3	4	5	6
2	3	4	2	3	2	3

0	1	2	3	4	5	6
?	x	?	?	?	?	?

Case 2: **Read from CSC**



CSC (in-edges)

Row Array

0	1	2	3	4	5
0	0	0	3	6	7

Column Array

0	1	2	3	4	5	6
0	1	3	0	1	4	0

Edge Values

0	1	2	3	4	5	6
?	?	?	Y	?	?	?

Incurs a random read

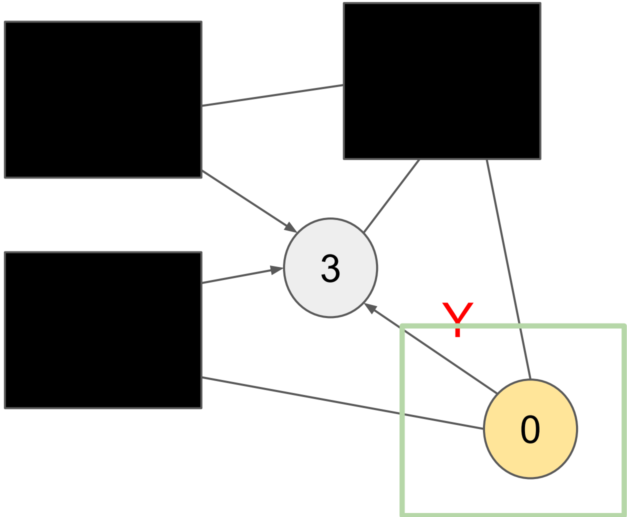
CSR (out-edges)

0	1	2	3	4	5
0	3	5	5	6	7

0	1	2	3	4	5	6
2	3	4	2	3	2	3

0	1	2	3	4	5	6
?	x	?	?	?	?	?

Case 2: **Read from CSC**



CSC (in-edges)

Row Array

0	1	2	3	4	5
0	0	0	3	6	7

Column Array

0	1	2	3	4	5	6
0	1	3	0	1	4	0

Edge Values

0	1	2	3	4	5	6
?	?	?	Y	?	?	?

Incurs a random read

CSR (out-edges)

0	1	2	3	4	5
0	3	5	5	6	7

0	1	2	3	4	5	6
2	3	4	2	3	2	3

0	1	2	3	4	5	6
?	Y	?	?	?	?	?

CSR + CSC

- Worst case scenario either we either have O(|E|) random reads or O(|E|) random writes.

Can we do better?

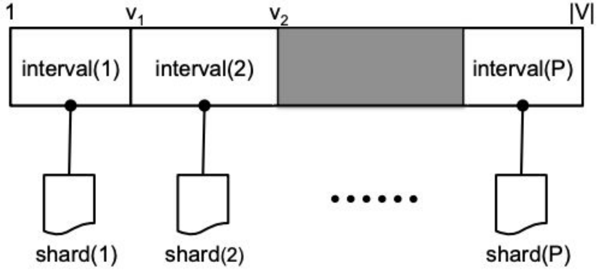
- SSD as a memory extension
- Exploiting locality
- Graph compression

Parallel Sliding Windows

STEPS

1. Load a subgraph into disk (so that we don’t get a bunch of random misses)
2. Update vertices and edges
3. Write updated values to disk
4. Profit

Parallel Sliding Windows - Storage Structure



- We have a graph G = (V, E)
- We split V into P disjoint partitions called **intervals.**
- For each interval, we have a shard (set of edges going into our partition). - Sorted by the source node.
- We choose P so that shards are of relative size and each shard can be loaded completely into memory (we want the max shard size to be a quarter of main memory)

How are graphs processed?

- GraphChi processes one **interval** at a time (as a subgraph)
- To actually create a subgraph we need to load our vertices, edges and our edge values
- Then we load the shard of our interval [edges going into our partition]
- How do we access the edges going **out** of our partition?

Accessing Out Edges of an Interval

- We keep track of P-1 sliding windows of the shards of the other P-1 intervals.

- Because we need to access a portion of each of our P shards to get the out edges of our interval of interest, we need P random reads

- Note, that if the degree distribution of a graph is not uniform, the window length is variable.

Parallel Sliding Windows - Storage Structure

Shard 1

src	dst	value
1		
	2	0.3
3		
	2	0.2
4		
	1	1.4
5		
	1	0.5
	2	0.6
6		
	2	0.8

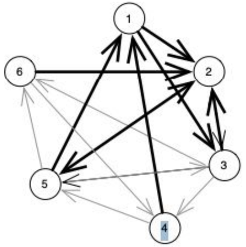
Shard 2

src	dst	value
1		
	3	0.4
2		
	3	0.3
3		
	4	0.8
5		
	3	0.2
6		
	4	1.9

Shard 3

src	dst	value
2		
	5	0.6
3		
	5	0.9
	6	1.2
4		
	5	0.3
5		
	6	1.1

(a) Execution interval (vertices 1-2)



(b) Execution interval (vertices 1-2)

Shard 1

src	dst	value
1		
	2	0.273
3		
	2	0.22
4		
	1	1.54
5		
	1	0.55
	2	0.66
6		
	2	0.88

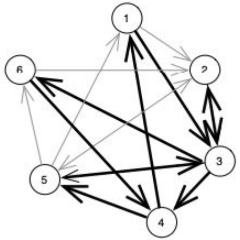
Shard 2

src	dst	value
1		
	3	0.364
2		
	3	0.273
3		
	4	0.8
5		
	3	0.2
6		
	4	1.9

Shard 3

src	dst	value
2		
	5	0.545
3		
	5	0.9
	6	1.2
4		
	5	0.3
5		
	6	1.1

(c) Execution interval (vertices 3-4)



(d) Execution interval (vertices 3-4)

1	2	3	4	5	6

Parallel Execution

- Execute our user defined function for each vertex in our **current interval** in parallel.

- Vertices that have edges with both end-points in the same interval are flagged as critical, and are updated in sequential order.

- Non-critical vertices do not share edges with other vertices in the interval, and can be updated safely in parallel.

Still maintains asynchronous execution … but limits parallelism

Updating values in disk

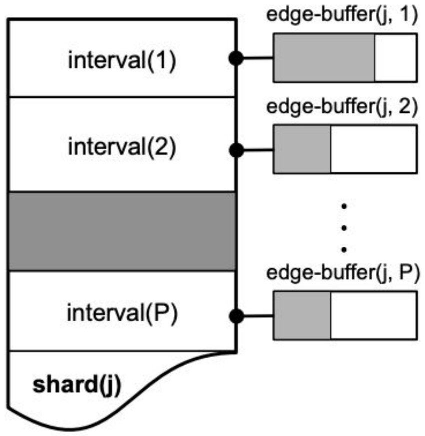
- Edge values are rewritten back to disk so that nodes in the next execution interval can access them.

- The entire memory shard is rewritten
- Only the sliding window of every other shard is rewritten

Number of non sequential writes is P (same as reads)

Handling Evolving Graphs

- Divide the shard into P logical parts: part j contains edges with source in the interval j.
- edge-buffer(p, j) for each logical part j, of shard p.
- When a new interval of vertices is loaded from disk, the edges in the edge-buffers are added to the in-memory graph



I/O Analysis

- I/O model -> Movement of data is more expensive than computation itself
- If both endpoints of an edge belong to the same vertex interval, the edge is read only once from disk; otherwise, it is read twice.
- If edges in both directions are modified, the number of writes is 2 per edge; if in only one direction, the number of writes is half as many.

$$\frac{2|E|}{B} \leq Q_B(E) \leq \frac{4|E|}{B} + \Theta(P^2)$$

Implementation

- Sharder -> Counts in-degree of vertices, computes a prefix-sum, divides vertices into P intervals
- Calculates memory needs using degreefiles
- Sub-intervals are used to account for unbalanced number of out-edges
- Selective Scheduling

Use Cases

- PageRank (transmit ranks through edges)
- Collaborative Filtering (recommend products based on purchases of others)
- Belief Propagation

Experimental Results

Most of the experiments were performed on a Apple Mac Mini computer (“Mac Mini”), with dual-core 2.5 GHz Intel i5 processor, 8 GB of main memory and a standard 256GB SSD drive (price $1,683 (Jan, 2012))

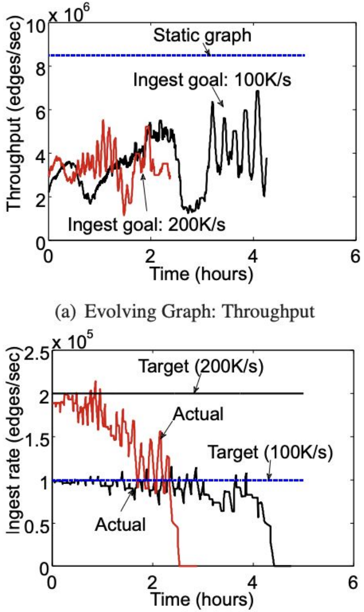
For experiments with multiple hard drives we used an older 8-core server with four AMD Opteron 8384 processors, 64GB of RAM, running Linux (“AMD Server”).

Experimental Results

Application & Graph	Iter.	Comparative result	GraphChi (Mac Mini)	Ref
Pagerank & domain	3	GraphLab[30] on AMD server (8 CPUs) **87 s**	**132 s**	-
Pagerank & twitter-2010	5	Spark [45] with 50 nodes (100 CPUs): **486.6 s**	**790 s**	[38]
Pagerank & V=105M, E=3.7B	100	Stanford GPS, 30 EC2 nodes (60 virt. cores), **144 min**	approx. **581 min**	[37]
Pagerank & V=1.0B, E=18.5B	1	Piccolo, 100 EC2 instances (200 cores) **70 s**	approx. **26 min**	[36]
Webgraph-BP & yahoo-web	1	Pegasus (Hadoop) on 100 machines: **22 min**	**27 min**	[22]
ALS & netflix-mm, D=20	10	GraphLab on AMD server: **4.7 min**	**9.8 min** (in-mem) **40 min** (edge-repl.)	[30]
Triangle-count & twitter-2010	-	Hadoop, 1636 nodes: **423 min**	**60 min**	[39]
Pagerank & twitter-2010	1	PowerGraph, 64 x 8 cores: **3.6 s**	**158 s**	[20]
Triange-count & twitter- 2010	-	PowerGraph, 64 x 8 cores: **1.5 min**	**60 min**	[20]

Table 2: **Comparative performance.** Table shows a selection of recent running time reports from the literature.

Experimental Results



(b) Ingest rate

Figure 9: (a,b) Evolving graphs: Performance when *twitter-2010* graph is ingested with a cap of 100K or 200K edges/sec, while simultaneously computing Pagerank.

Related Work

An improved memory management scheme for large scale graph computing engine GraphChi by Yifang Jiang et al. (2014/2015)

GraphMP: An Efficient Semi-External-Memory Big Graph Processing System on a Single Machine (2017)

GraphLab: A New Framework For Parallel Machine Learning

PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

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

TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC

Strengths, Weaknesses, and Next Steps

Strengths

- Easy to understand, simple
- Good system to use if you know nothing about distributed computing/don't have access to many resources
- Maintains the vertex-centric model
- Open source implementation

Weaknesses

- Did not go much into depth on selective scheduling and how critical nodes are scheduled.

Next Steps

- Introduce parallel shard streaming to fully utilize multiple SSD channels.
- Combine with edge compression/decompression on the fly (like GraphMP).
- Edge cache for frequently used shards
- Store vertices in memory (semi-external)

Discussion Questions

- Realistically, what are some good use cases for data mining large graphs on a single PC
- If you own a big graph, wouldn’t you want the fastest way to process it (distributed vs singular node)
- Is there still value in single-machine graph processing in the era of cheap cloud clusters and GPUs?
- Since we don’t have explicit message passing, is this model less expressive (do some algorithms have more affinity for the Pregel-like model)