Authors: Aapo Kyrola, Guy Blelloch, Carlos Guestrin

Presenter: Terryn Brunelle

Motivation

Motivation

- Large graphs require distributed computing
  - Ex: Social networks, web graphs, protein interaction graphs
- Distributed graph algorithm development challenging to non-experts

Existing Work: Vertex-Centric

Algorithm 1: Typical vertex **update-function**

```
1 Update(vertex) begin
2     x[] ← read values of in- and out-edges of vertex ;
3     vertex.value ← f(x[]) ;
4     foreach edge of vertex do
5         edge.value ← g(vertex.value, edge.value);
6     end
7 end
```

Problems

- Can scale to billions of edges by distributing computation…
- But to do so need to partition graph across cluster nodes
- Finding efficient graph cuts is difficult

Goal

Find graph cuts that

- Minimize communication between nodes
- Are balanced

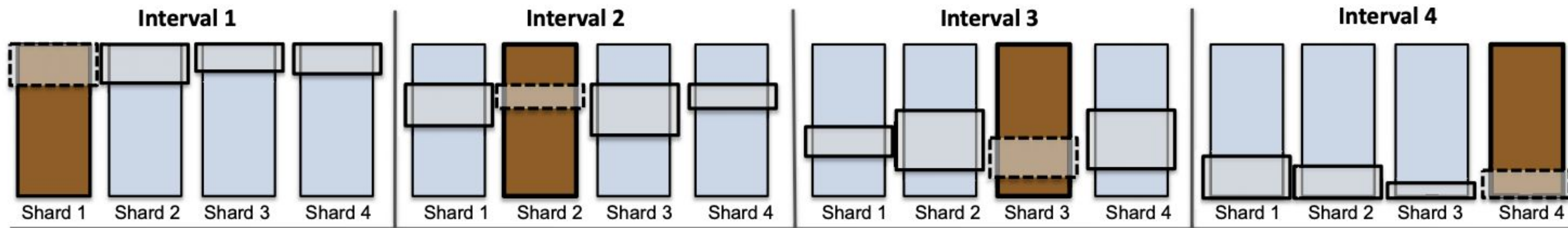
GraphChi

Parallel Sliding Windows (PSW)

- Process very large graphs on disk
- Asynchronous model of computation

1. Load subgraph from disk
2. Update vertices and edges
3. Write updated values to disk

1. Load Subgraph from Disk



- Within each interval
    - Execute update-function for each vertex in parallel
- Enforce external determinism
    - Critical vertices updated in sequential order
    - Non-critical vertices updated in parallel

3. Write Updated Values to Disk

- Load edges from disk in large blocks cached in memory
- Write to blocks and load them back to disk to replace old data
  - Completely rewrite memory shard
  - Only rewrite active sliding windows of other shards
- P non-sequential disk writes per interval

PSW Example

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

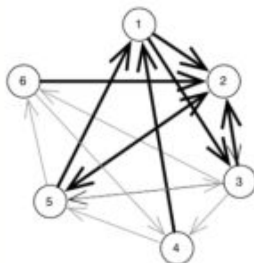
Shard 2

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

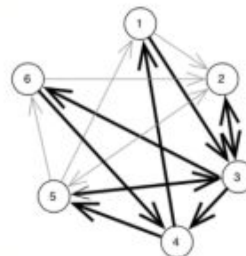
Shard 2

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)

I/O Complexity

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

GraphChi Data Pre-Processing

Compact Shard Format

- Adjacency shard -- edge array for each vertex
- Edge data shard -- flat array of edge values

Sharder

- Count vertex in-degrees
- Compute prefix sum over degree array
- Divide vertices into P intervals
- Write each edge to temporary scratch file of owning shard
- For each file, sort edges and write them in compact format

I/O Cost: 5|E|/B + |V|/B

GraphChi Implementation

- Calculate exact memory needed for execution interval
  - Use multithreading to access needed vertices
  - Degreefile stores in/out degrees for each vertex
- Divide execution into sub-intervals
- Evolving graphs
- Selective Scheduling

Main Execution

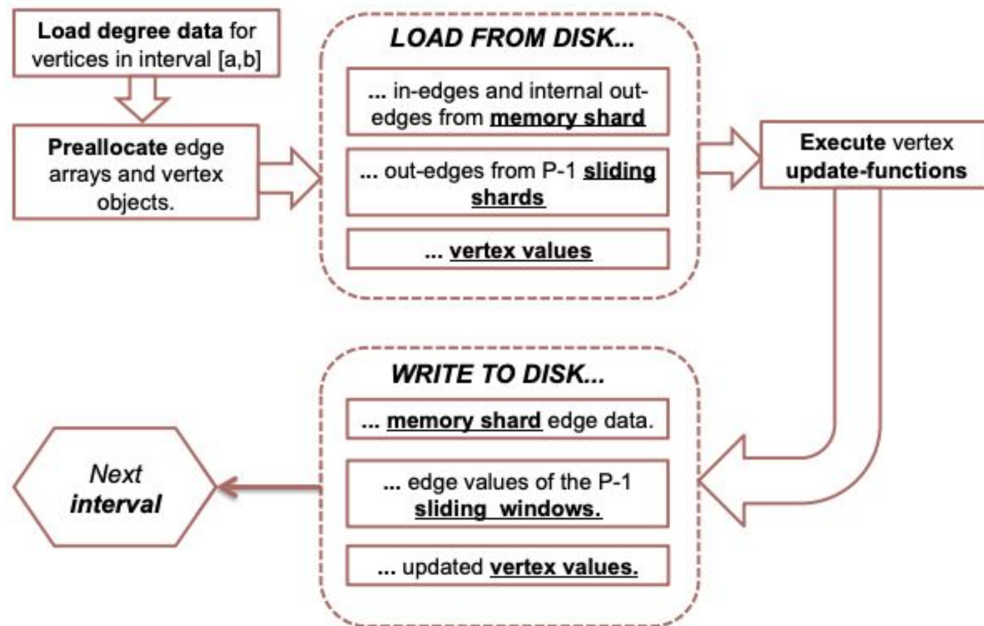


Figure 5: **Main execution flow.** Sequence of operations for processing one execution interval with GraphChi.

Programming Model

Algorithm 4: Pseudo-code of the vertex update-function for weighted PageRank.

```
1 typedef: VertexType float
2 Update(vertex) begin
3     var sum ← 0
4     for e in vertex.inEdges() do
5         sum += e.weight * neighborRank(e)
6     end
7     vertex.setValue(0.15 + 0.85 * sum)
8     broadcast(vertex)
9 end
```

Standard Model

Algorithm 5: Type definitions, and implementations of neighborRank() and broadcast() in the standard model.

```
1 typedef: EdgeType { float weight, neighbor_rank; }
2 neighborRank(edge) begin
3 |    return edge.weight * edge.neighbor_rank
4 end
5 broadcast(vertex) begin
6 |    for e in vertex.outEdges() do
7 |    |    e.neighbor_rank = vertex.getValue()
8 |    end
9 end
```

In-Memory Model

Algorithm 6: Datatypes and implementations of neighborRank() and broadcast() in the alternative model.

```
1 typedef: EdgeType { float weight; }
2 float[] in_mem_vert
3 neighborRank(edge) begin
4 |    return edge.weight * in_mem_vert[edge.vertex_id]
5 end
6 broadcast(vertex) /* No-op */
```

Applications

- SpMV Kernels
- Graph Mining
- Collaborative Filtering
- Probabilistic Graphical Models

Experimental Results

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]

Experimental Results

- Within constant factor of other systems
- Uses a fraction of the resources
- Can process 5-20 million edges/second on Mac Mini

Conclusion

Strengths/Weaknesses

Strengths

- Sparse graphs
- Efficient on consumer PC
- Makes large-scale graph computation widely accessible

Weaknesses

- Difficult to benchmark results
- Dynamic ordering and graph traversals

Directions for Future Work

- Evaluating more efficient shard formats
- Testing on additional infrastructures

Discussion Questions

- Even though there were no comparable models to benchmark GraphChi against, do you find the experimental results compelling?
- How would GraphChi perform on dense graphs?
- Could GraphChi be adapted to support graph traversal problems?