GraphChi: Large-Scale Graph Computation on Just a PC

By: Aapo Kyrola, Guy Blelloch, and Carlos Guestrin

Reviewed By: Jonathan Li

Previous Graph Systems and Frameworks

Were designed for large clusters of computers or at least a large number of cores.

What happens when you want to run algorithms on large graphs with billions of edges on a normal PC?

- There simply isn’t enough memory on the computer to store the entire graph.
- Solution: Store the graph on disk instead.
  - Problem: May be I/O-inefficient if many random reads or writes required

Contributions in this Paper

1. Parallel Sliding Windows (PSW) method
    a. Loading subgraph from disk
    b. Updates vertex and edge values
    c. Writes updates back to disk
2. The GraphChi system design

PSW (Subgraphs Structure)

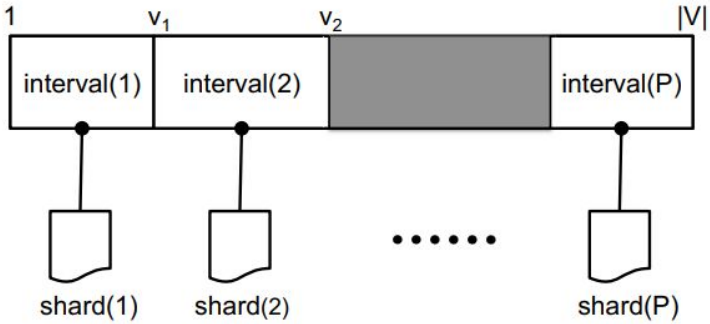
Vertices are divided into P intervals, each interval has a shard associated with it.

A shard consists of all edges that have destination in the interval.

Edges are sorted by source vertex.

Intervals are chosen to balance the size of shards.

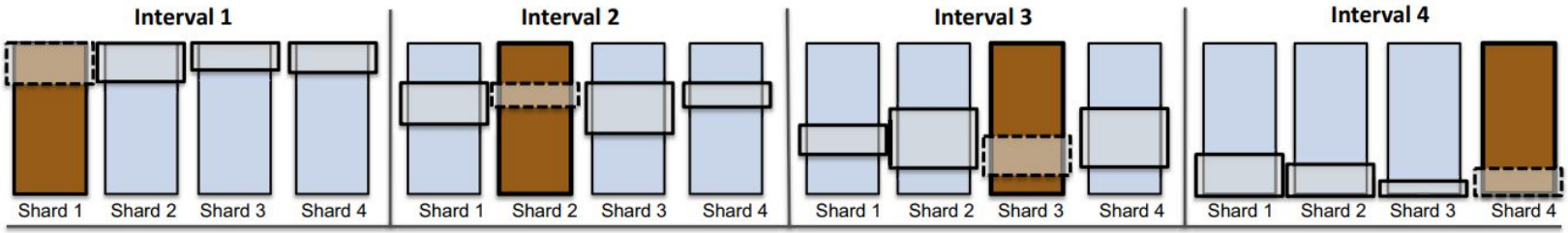
Number of intervals is chosen to ensure any shard can be loaded into memory.



PSW (Loading Subgraphs)

PSW runs by iterating through all intervals.

For interval p, Shard(p) is loaded fully into memory. Then, we iterate through all other shards and load in edges with source in the interval.



PSW (Parallel Updates)

Critical edges (edges with both endpoints in the interval) are updated sequentially.

Non-critical edges are updated in parallel.

Shard 1

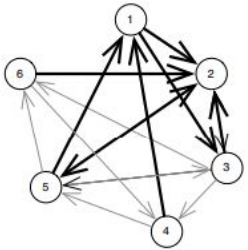
src	dst	value
1		
	2	0.3
3		
	2	0.2
4		
	1	1.4
5		
	1	0.5
	2	0.6
6		
	2	0.8

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



Shard 1

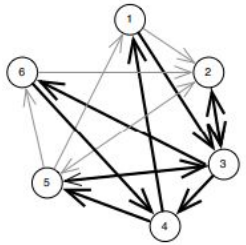
src	dst	value
1		
	2	0.273
3		
	2	0.22
4		
	1	1.54
5		
	1	0.55
	2	0.66
6		
	2	0.88

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



PSW (Update to Disk)

After finishing updates to all vertices and edges, modified blocks are written back to disk.

Next iteration of PSW will use new values for vertices and edges.

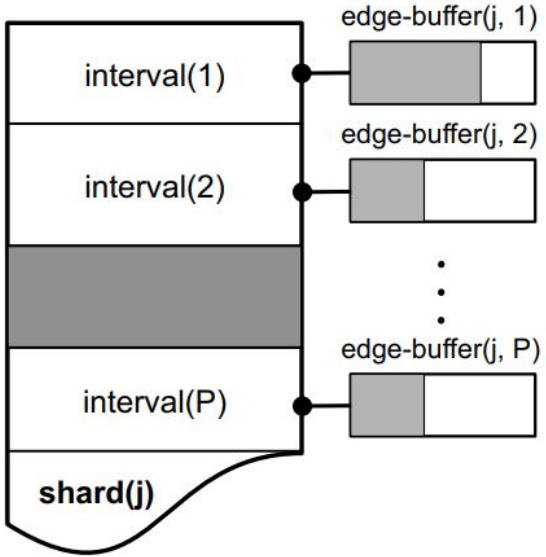
PSW (Mutable Graph)

Each shard can be split into P parts, where the p^{th} part contains all edges with source in the p^{th} interval.

Each of these parts has an edge buffer, and additional edges are first added to the edge buffer.

Edges are loaded in when the interval is loaded.

Otherwise, we check at end of iteration whether the buffer or shard has exceeded a predetermined limit.



I/O-cost of PSW

The I/O-cost of PSW for each execution interval is bounded by

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

For each execution interval, there are P non-sequential disk seeks across P-1 intervals, which means there are $O(P^2)$ disk seeks per iteration.

Each edges is read twice (or once if critical) and if updated, is also written twice.

System Design and Implementation (Shards)

Shard is formatted so that each vertex in the interval has an edge array.

Each edge array begins with the length of the array followed by a list of the neighbors.

If a vertex has no neighbors, we instead write a zero length byte followed by another value indicating how many subsequent vertices have no neighbors.

The edge data/values are contained in a flat array of user-defined type.

Sharder (Preprocessing)

GraphChi includes a program for creating shards from standard graph file formats.

1. Counts the in-degree of each vertex, then computes the prefix sum of the array to split the vertices into P intervals.
2. Does another pass of edges, writing each edge to a scratch file associated to each shard.
3. Sorts the edges in each scratch file and compresses the data.
4. Also compiles a degree file which contains the in-degree and out-degree for each vertex.

Main Execution

Uses the degree file to calculate exactly how much memory is needed to store the edge arrays before initializing a subgraph.

Vertex values are also stored in a flat array in sequential order of user-defined type.

In cases where there are a lot of vertices with high out-degree in a single interval, the interval is split into sub-intervals, and each sub-interval is run in a separate iteration.

When adding vertices, the degree file needs to be updated and vertex data file will also grow. In addition, number of intervals must remain consistent with the number of shards.

Pagerank Example

Unlike other programs such as GraphLab, GraphChi does not allow vertices to modify the values of their neighbors (not necessarily loaded into memory).

Thus, in GraphChi, vertices broadcast neighbor updates via edges.

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
```

Other Applications

GraphChi has been implemented with:

- Sparse Matrix Vector Multiplication
- Pagerank
- Connected Components
- Community Detection
- Triangle Counting
- Collaborative Filtering
- Probabilistic Graph Models

Experimental Setup

Apple Mac Mini Computer

- Dual-core 2.5 GHz Intel i5 Processor
- 8 GB Main Memory
- 256 GB SSD Drive
- 750 GB, 7200 rpm Hard Drive

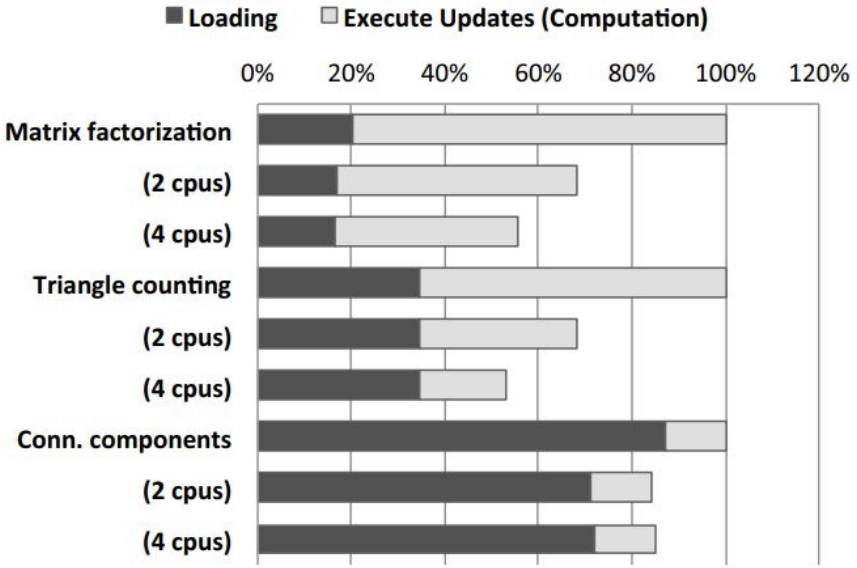
Preprocessing Results

Graph name	Vertices	Edges	P	Preproc.
live-journal [3]	4.8M	69M	3	0.5 min
netflix [6]	0.5M	99M	20	1 min
domain [44]	26M	0.37B	20	2 min
twitter-2010 [26]	42M	1.5B	20	10 min
uk-2007-05 [11]	106M	3.7B	40	31 min
uk-union [11]	133M	5.4B	50	33 min
yahoo-web [44]	1.4B	6.6B	50	37 min

Algorithm Results

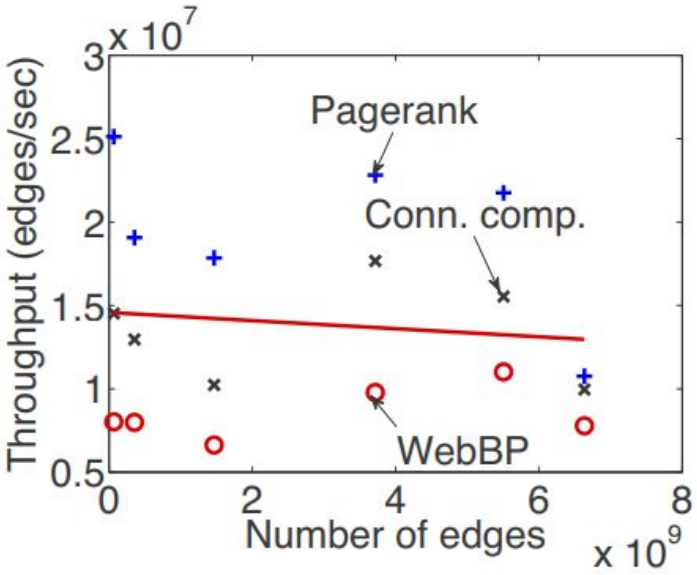
Application & Graph	Iter.	Comparative result	GraphChi (Mac Mini)	Ref
Pagerank & domain	3	GraphLab[30] on AMD server (8 CPUs) **87 s**	**132 s**	-
Pagerank & twitter-2010	5	Spark [45] with 50 nodes (100 CPUs): **486.6 s**	**790 s**	[38]
Pagerank & V=105M, E=3.7B	100	Stanford GPS, 30 EC2 nodes (60 virt. 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]

Parallelism and Overhead

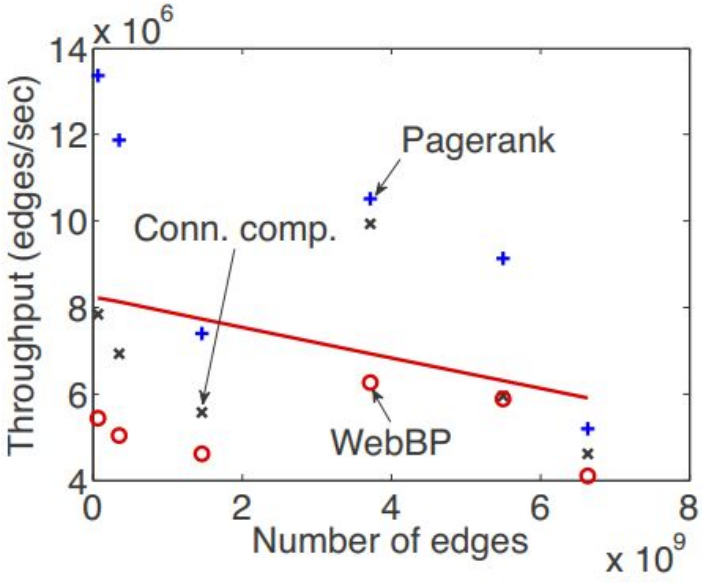


Application	SSD	In-mem	Ratio
Connected components	45 s	18 s	2.5x
Community detection	110 s	46 s	2.4x
Matrix fact. (D=5, 5 iter)	114 s	65 s	1.8x
Matrix fact. (D=20, 5 iter.)	560 s	500 s	1.1x

SSD vs Hard Drive



(a) Performance: SSD



(b) Performance : Hard drive

Conclusion

Main result of the paper was introducing the Parallel Sliding Windows method and the GraphChi implementation.

This method can be used on consumer PCs to run algorithms previously restricted to mainly cluster computing.