GraphChi: Large-Scale Graph Computation on Just a PC

By: Aapo Kyrola (Carnegie Mellon University), Guy Blelloch (Carnegie Mellon University), Carlos Guestrin (University of Washington)

Presentation by: Jessica Zhu

Motivations

- Real-world graphs are huge
- Computation on these graphs is very expensive and time-consuming
- Distributed graph algorithms are hard to understand

Contributions

- Parallel Sliding Windows (PSW)
  - Small number of non-sequential accesses to disk
  - Implements asynchronous model of computation
  - Processes large graphs from disk with theoretical guarantees
- GraphChi
  - Design, evaluation, and implementation in C++
  - Able to solve problems previously only solvable on cluster computing

Disk-Based Graph Computation

- Existing models are 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
```

Disk-Based Graph Computation

- Existing models use the Bulk-Synchronous Parallel (BSP) model
  - Update functions use values from previous iteration
  - Simple to implement, allows maximum parallelization
  - Synchronization steps (after each iteration) are expensive

- Asynchronous model
  - Update functions use most recent values of edges and vertices
  - Ordering of updates is dynamic
  - Converges in situations where BSP does not

Disk-Based Graph Computation

- Compressed Sparse Row and Compressed Sparse Column storage
- Modifying the value of a vertex
  - New value must be read from set of out-edges (random read) OR
  - New value is written to in-edge list (random write)

- Possible Solutions
  - SSD as a memory extension: can’t handle accessing millions of edges per second
  - Exploiting locality: unpredictable, depends highly on structure of graph
  - Graph compression: doesn’t work if data is stored with the nodes and edges

Parallel Sliding Windows (PSW)

- Loads subgraph from disk
- Updates vertices and edges
- Writes updated values to disk

PSW: Loading subgraph from disk

- Vertices V are split into P disjoint intervals
- Each interval has a shard that stores all edges going into the interval
- Edges are stored in order of their source
- Intervals balances number of edges in each shard
- Does graph computation in execution intervals
- First load shard(p) into memory, call it memory-shard
- Out-edges are stored in consecutive chunks in the other shards, requiring P-1 block reads
- Edges for interval(p+1) are stored immediately after interval(p)
- When PSW moves onto the next interval, it slides over window, other shards are called sliding shards
- Window length is variable if degree distribution is not uniform

PSW: Loading subgraph from disk

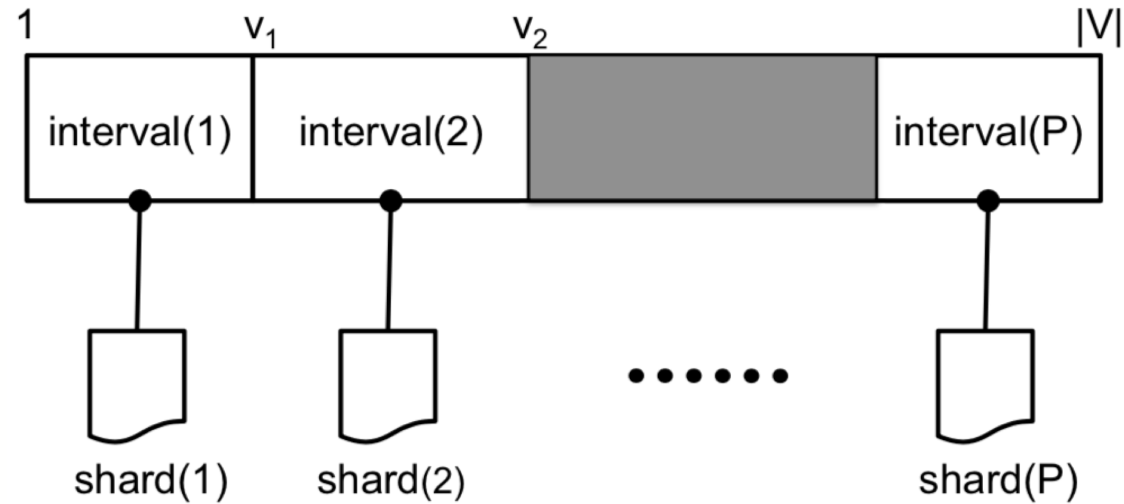


Figure 1: The vertices of graph (V, E) are divided into P intervals. Each interval is associated with a shard, which stores all edges that have destination vertex in that interval.

PSW: Updating vertices and edges

- Subgraph for interval p has been loaded to disk
- Call update-function for each vertex in parallel
- External determinism prevents race conditions (accessing edges concurrently), guarantees each run of PSW produces same result
  - To implement: vertices with end-points of edges in the same interval are marked as critical and executed sequentially (in line with the asynchronous model)

PSW: Updating vertices and edges

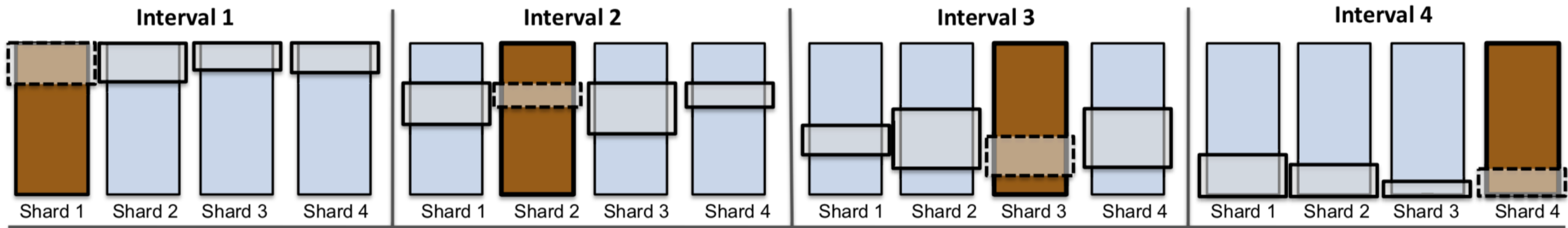


Figure 2: Visualization of the stages of one iteration of the Parallel Sliding Windows method. In this example, vertices are divided into four intervals, each associated with a shard. The computation proceeds by constructing a subgraph of vertices one interval a time. In-edges for the vertices are read from the **memory-shard** (in dark color) while out-edges are read from each of the **sliding shards**. The current **sliding window** is pictured on top of each shard.

PSW: Updating vertices and edges

Algorithm 2: Parallel Sliding Windows (PSW)

```
1  foreach iteration do
2      shards[] ← InitializeShards(P)
3      for interval ← 1 to P do
4          /* Load subgraph for interval, using Alg. 3. Note,
           that the edge values are stored as pointers to the
           loaded file blocks. */
5          subgraph ← LoadSubgraph (interval)
6          parallel foreach vertex ∈ subgraph.vertex do
7              /* Execute user-defined update function,
8               which can modify the values of the edges */
9              UDF_updateVertex (vertex)
10         end
11         /* Update memory-shard to disk */
12         shards[interval].UpdateFully()
13         /* Update sliding windows on disk */ for
           s ∈ 1, .., P, s ≠ interval do
14             shards[s].UpdateLastWindowToDisk()
15         end
16     end
17 end
```

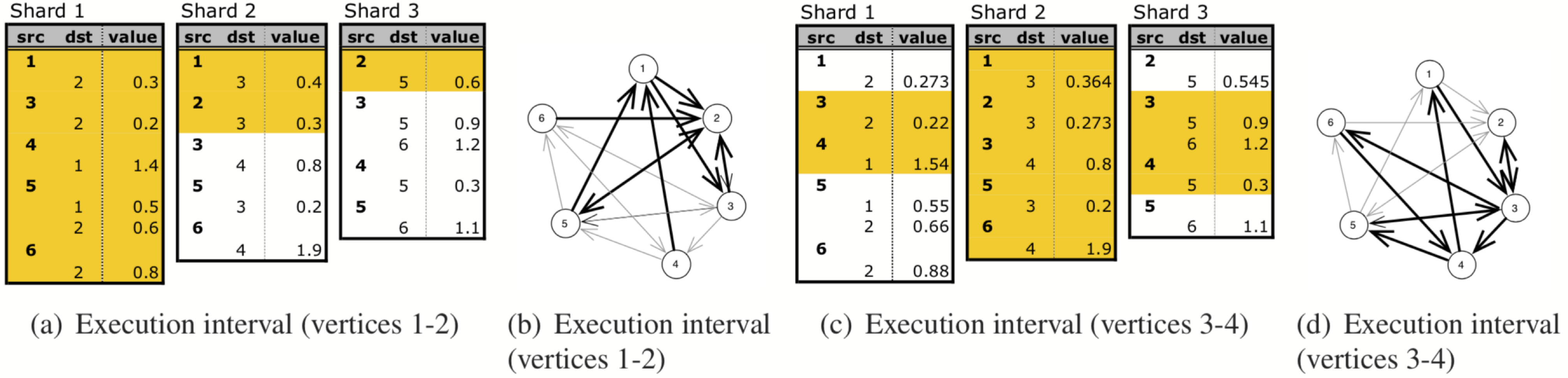
PSW: Writing updated values to disk

- Edges are loaded from disk in large blocks which are cached in memory
- Modifications directly modify blocks themselves, PSW overwrites old data when it updates
- Active sliding window is rewritten to disk
- Number of non-sequential writes for an execution interval is P

Algorithm 3: Function LoadSubGraph(p)

```
   Input  : Interval index number p
   Result: Subgraph of vertices in the interval p
1  /* Initialization */
2  a ← interval[p].start
3  b ← interval[p].end
4  G ← InitializeSubgraph (a, b)
5  /* Load edges in memory-shard. */
6  edgesM ← shard[p].readFully()
7  /* Evolving graphs: Add edges from buffers. */
8  edgesM ← edgesM ∪ shard[p].edgebuffer[1..P]
9  foreach e ∈ edgesM do
10     /* Note: edge values are stored as pointers. */
11     G.vertex[edge.dest].addInEdge(e.source, &e.val)
12     if e.source ∈ [a, b] then
13         G.vertex[edge.source].addOutEdge(e.dest, &e.val)
14     end
15 end
16 /* Load out-edges in sliding shards. */
17 for s ∈ 1, .., P,  s ≠ p do
18     edgesS ← shard[s].readNextWindow(a, b)
19     /* Evolving graphs: Add edges from shard's buffer p */
20     edgesS ← edgesS ∪ shard[s].edgebuffer[p]
21     foreach e ∈ edgesS do
22         G.vertex[e.src].addOutEdge(e.dest, &e.val)
23     end
24 end
25 return G
```

PSW in action



(a) Execution interval (vertices 1-2) (b) Execution interval (vertices 1-2) (c) Execution interval (vertices 3-4) (d) Execution interval (vertices 3-4)

Figure 3: Illustration of the operation of the PSW method on a toy graph (See the text for description).

Evolving Graphs

- Support changes in graph structure
  - Allow adding edges to graphs
  - Allows removal of edges (flag them, delete when shard is rewritten to disk)
- Divide shard into P logical parts: part j contains edges with source in the interval j
  - Edge-buffer(p, j) is in-memory
  - When edge is added to graph, add it to corresponding edge-buffer
  - When interval is loaded from disk, edges from edge-buffers are added to in-memory graph
  - If number of edges in edge-buffers exceeds limit, write edges to disk

Evolving Graphs

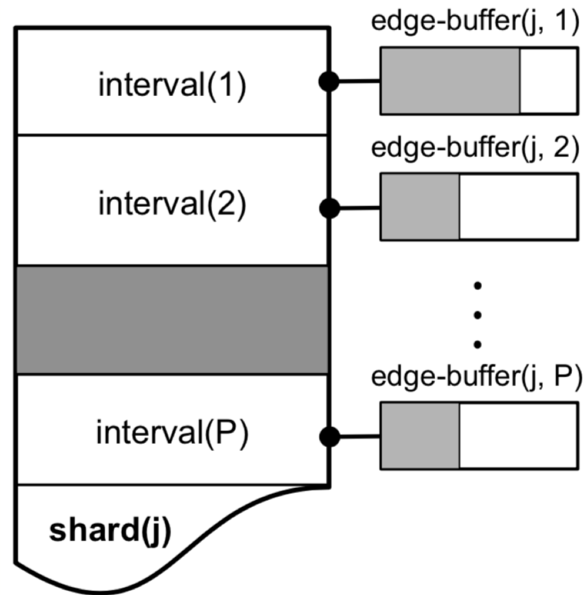


Figure 4: A shard can be split into P logical parts corresponding to the vertex intervals. Each part is associated with an in-memory edge-buffer, which stores the inserted edges that have not yet been merged into the shard.

I/O Complexity

- Cost = number of block transfers from disk to main memory
- B: size of block transfer
- Total data size = |E|, as each edge is stored once
- Shards have sizes |E|/P
- Each edge is accessed twice (once in each direction)
- Each edge is written once or twice (once if both endpoints of edge belong to same vertex interval)
- Often PSW requires P non-sequential disk seeks to load edges from the P-1 sliding shards for an execution interval

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

GraphChi System Design

- Shard Data Format
  - Fast to generate and read
  - Adjacency shard stores an edge array for each vertex in order
  - Edge shard data is a flat array of edge values in user defined type

GraphChi System Design: Preprocessing

- Sharder
  - Counts the in-degree of each vertex, computes prefix sum to divide graph into equal intervals (one pass)
  - Write each edge to a temporary file of the owning shard (one pass)
  - Process each temporary file to sort the edges and compress them
  - Compute a binary degree file with in and out degree of each vertex

- P is chosen so that the largest shard is at most ¼ size of available memory (other memory needed to store pointers, buffers, auxiliary data structures)
- Total cost: $\frac{5|E|}{B} + \frac{|V|}{B}$

GraphChi Implementation

- Efficient subgraph construction
  - Calculates the exact amount of memory needed to store and perform computation on an execution interval
  - Can do this using degreefile, which stores all in and out degrees of each vertex (using prefix sum, can calculate exactly how many edges they need to store)
  - I/O cost: $2\lceil |V|/B \rceil$
- Selective scheduling
  - Update can flag a neighboring vertex to be updated, typically if edge value changes significantly
  - Can be used to implement incremental computation: when an edge is created, its source or destination vertex is added to the schedule

GraphChi: Programming Model

- Adjacency shard: stores edge array for each vertex in order
- Edge data shard: flat array of edge values

- Sharder: handles preprocessing, which is I/O efficient and can be done with limited memory
  - Counts the in-degree of each vertex and calculates prefix sum to divide the graph into P equal intervals (one pass)
  - Write each edge to temporary file of owning shard (one pass)
  - Process each of these files to sort edges and write in compact format
  - Compute binary degreefile (both in and out edges) for every vertex

GraphChi: Execution

- Efficient subgraph construction
  - Calculate exact memory needed for an execution interval using degreefile
  - Use multithreading to access the vertices needed
- Sub intervals
  - Divide execution interval into sub intervals (some intervals may have lots of edges that don’t fit into memory)
  - Allows same shard files to be used with different amounts of memory, I/O costs not affected
- Evolving graphs
  - Keep track of changing degreefiles, vertex interval sizes
- Selective scheduling
  - Updates flag neighboring vertices to also be updated

GraphChi Implementation

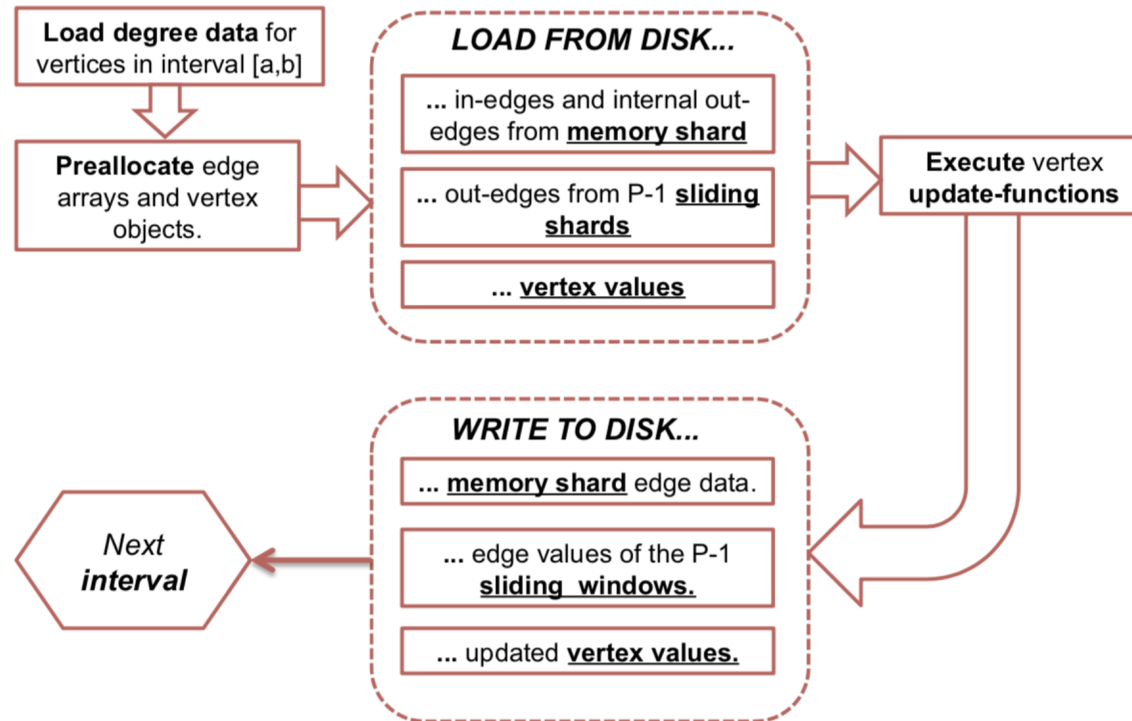


Figure 5: **Main execution flow.** Sequence of operations for processing one execution interval with GraphChi.

GraphChi: Programming Model

- Similar to programs for Pregel or GraphLab
  - Pregel uses messaging, GraphChi directly modifies vertices and edges
  - GraphLab directly reads and modifies neighboring vertices, GraphChi does not

GraphChi Implementation

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
```

GraphChi Implementation

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
```

GraphChi Implementation

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 */
```

GraphChi Applications

- SpMV Kernels, PageRank
- Graph Mining
- Collaborative Filtering
- Probabilistic Graphical Model

Experimental Setup

- Test Setup: Mac Mini with 8GB of main memory, 256GB SSD drive, 750GB hard drive + 8 core server with 64GB RAM

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

Table 1: Experiment graphs. Preprocessing (conversion to shards) was done on Mac Mini.

Experimental Results

- No direct models to compare against
- Runtimes are within a constant factor when compared to other distributed systems with more cores
- PowerGraph is a distributed version of GraphChi, can perform one iteration of PageRank on twitter-2010 in 5 seconds (GraphChi: 158s)

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.

Scalability and Performance

- Performance measured as throughput (number of edges processed in a second)
  - GraphChi can process 5-20million edges/s on Mac Mini
  - Using a hard drive for memory is sufficient, can be improved by adding more hard drives
  - Using different block sizes can change efficiency

Scalability and Performance

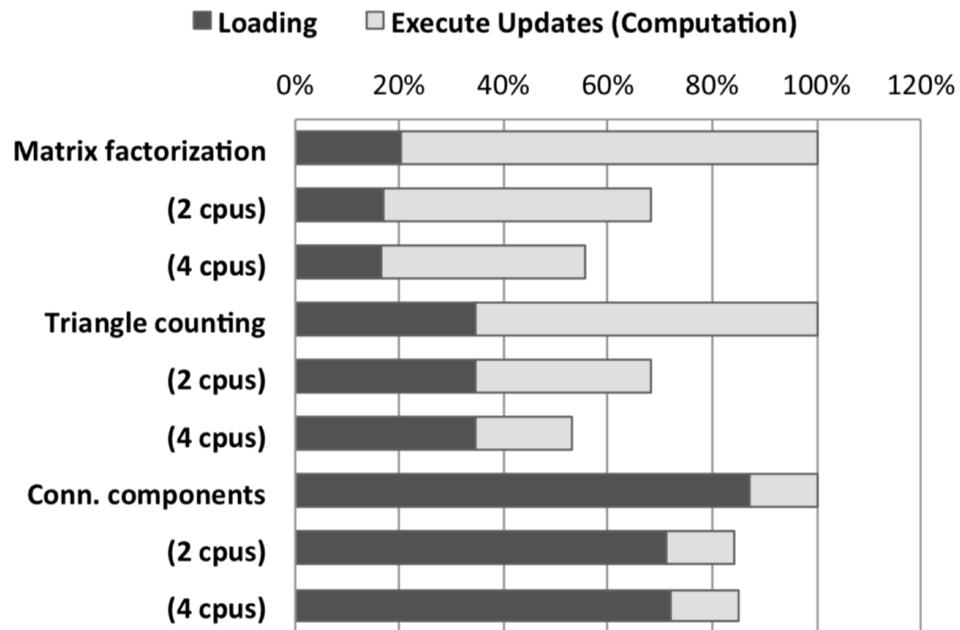


Figure 6: Relative runtime when varying the number of threads used used by GraphChi. Experiment was done on a MacBook Pro (mid-2012) with four cores.

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

Table 3: Relative performance of an in-memory version of GraphChi compared to the default SSD-based implementation on a selected set of applications, on a Mac Mini. Timings include the time to load the input from disk and write the output into a file.

Strengths and Weaknesses

- Paper was well organized and pseudocode helped with overall understanding of the content
- Some parts were repetitive, like the description of how the algorithm was the same as the description of GraphChi
- Results are promising, but no real benchmark to how “good” they are

Discussion Questions

- GraphChi is designed for sparse real-world graphs. Does it perform as well on dense graphs?
- How well does GraphChi perform with different graph algorithms (e.g. Bellman-Ford, Dijkstra's, etc.)?
- How does the number of computations/iterations necessary to run GraphChi compare with other graph computation algorithms?