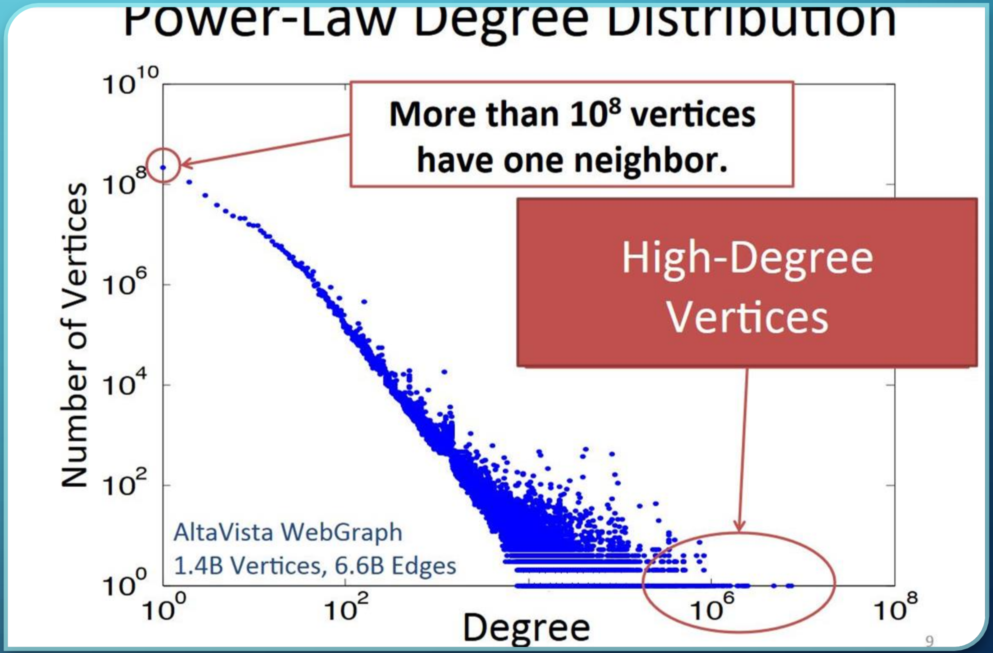
POWERGRAPH

BY KYLE SWEENEY

POWERGRAPH FROM GONZALES ET AL FROM CARNEGIE MELLON AND UNIVERSITY OF WASHINGTON

BACKGROUND – THE PROBLEM

- Natural graphs are everywhere, from Facebook, to Twitter, to Netflix to Genomics.
- Natural Graphs suffer from Power-Law degree Distribution



BACKGROUND – DISTRIBUTION PROBLEM

- Power-Law graphs are very difficult to partition well, as they do not have low-cost balanced cuts
- A common abstraction is the Graph-Parallel Abstraction
  - Users provide vertex-programs which will run on vertexes simultaneously and communicate with one another
  - This leads to really high communication overheads on systems like Pregel, GraphLab, etc

SOLUTION - POWERGRAPH

- GAS Decomposition Gather, Apply, Scatter
  - You make vertices with really high-degrees parallel
- Vertex Partitioning
  - If you perform a vertex cut, that is, split on high-degree vertices, then you can reduce information
  - Take really high degree vertices, and spread them across the network

GATHER, APPLY, SCATTER

- Each are a user defined function applied on the high-degree vertexes
- Gather
  - Happens on HDV + neighbor
  - Eventually you Add up the results
- Apply
  - Happens on the HDV, take the result from Gather
- Scatter
  - Let everyone else know HDV updated info, on edges and verticies

HOW TO DO VERTEX CUTS

- Assign edges to machines evenly
- Assign edges as they are loaded
- Placement Strategies
  - Random Placement
  - Coordiated greedy placement
  - Oblivious greedy

PLACEMENT STRATEGIES GREEDY HEURISTICS

- $\arg\min_k E\left[\sum_{v \in V} |A(v)| \,\middle|\, A_i, A(e_{i+i}) = k\right]$
  - A_i is the assignment for the previous i edges.
- Coordinated
  - Use a distributed table to maintain values for $A_i(v)$. Each machine then runs and updates the table everyone sees. Use local caching
- Oblivious
  - Do coordinated, but don’t coordinate

OTHER COOL FEATURES

- Execution Models Synch, Async, Async + Serializable
- Delta caching aka partial sums caching

EXECUTION MODEL

- Synchronous
  - Applies Gather, apply, scatter in order, with each phase having a barrier to keep every computer in step
  - Changes made at the end of each step are applied everywhere and visible
  - Not exactly efficient, thanks to barriers
- Async
  - Runs verticies when cores and network becomes available.
  - Data changes happen when they do, and are visible after the change to neighbors
- Serializable
  - All parallel execution has corresponding sequential
  - Prevents adjacent vertex-programs from running at the same time

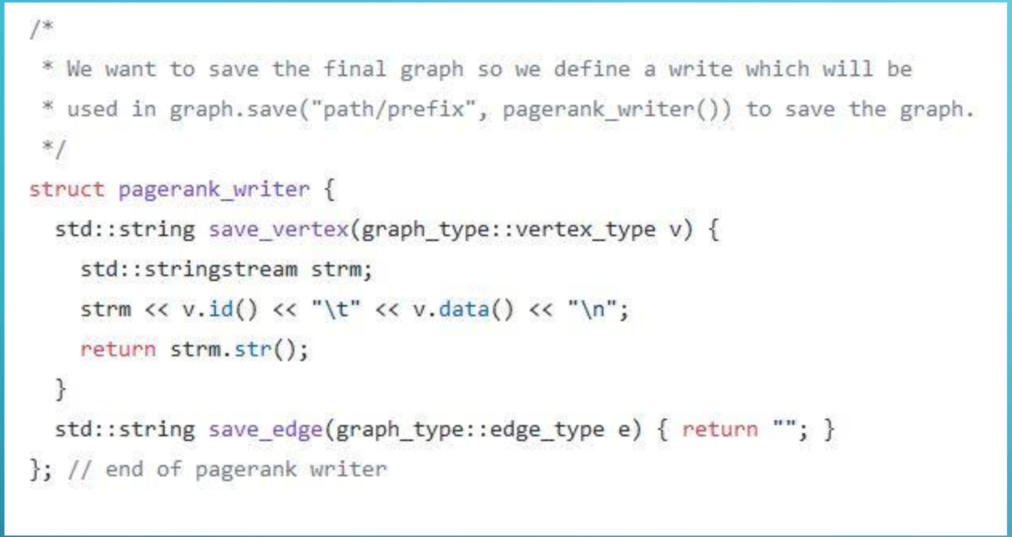
HOW ARE GRAPHS EXPRESSED?

- You can write your own parser to represent graphs
- They by default accept graphs generally written as DAGs
- They don’t give away too much info about the internal workings….

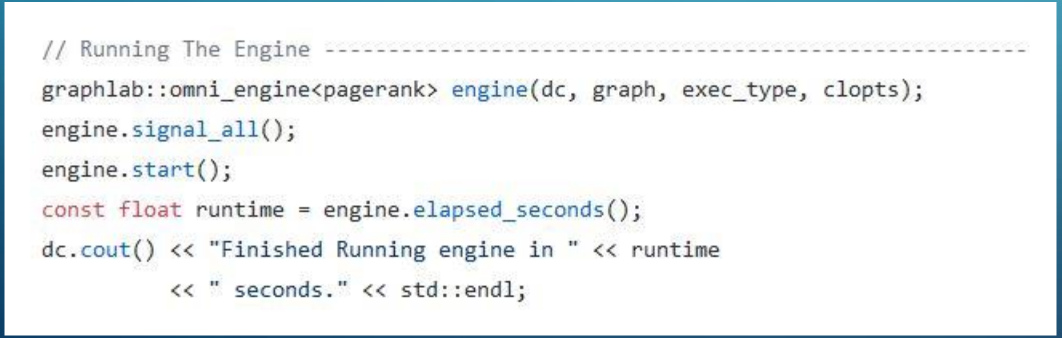
SOFTWARE FUNDAMENTALS

- Requires A whole mess of Librarires
  - Fundamentally a C++ MPI library
  - Boost, zlib, patch, JDK (for hdfs support), libevent, libjson, tcmalloc (optional)

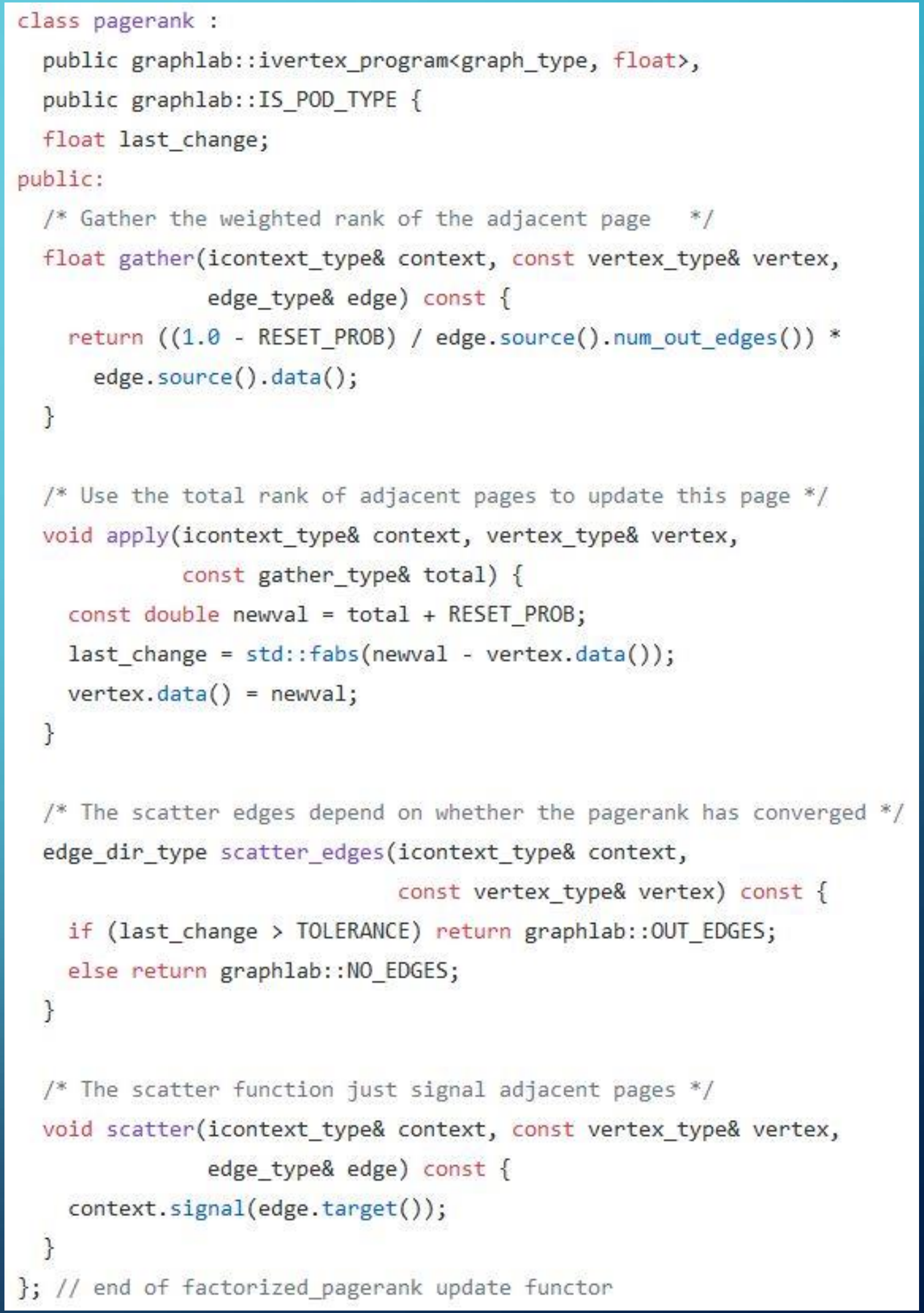
SAMPLE PAGE-RANK PROGRAM



```
/*
 * We want to save the final graph so we define a write which will be
 * used in graph.save("path/prefix", pagerank_writer()) to save the graph.
 */
struct pagerank_writer {
  std::string save_vertex(graph_type::vertex_type v) {
    std::stringstream strm;
    strm << v.id() << "\t" << v.data() << "\n";
    return strm.str();
  }
  std::string save_edge(graph_type::edge_type e) { return ""; }
}; // end of pagerank writer
```



```
// Running The Engine -------------------------------------------------------
graphlab::omni_engine<pagerank> engine(dc, graph, exec_type, clopts);
engine.signal_all();
engine.start();
const float runtime = engine.elapsed_seconds();
dc.cout() << "Finished Running engine in " << runtime
          << " seconds." << std::endl;
```



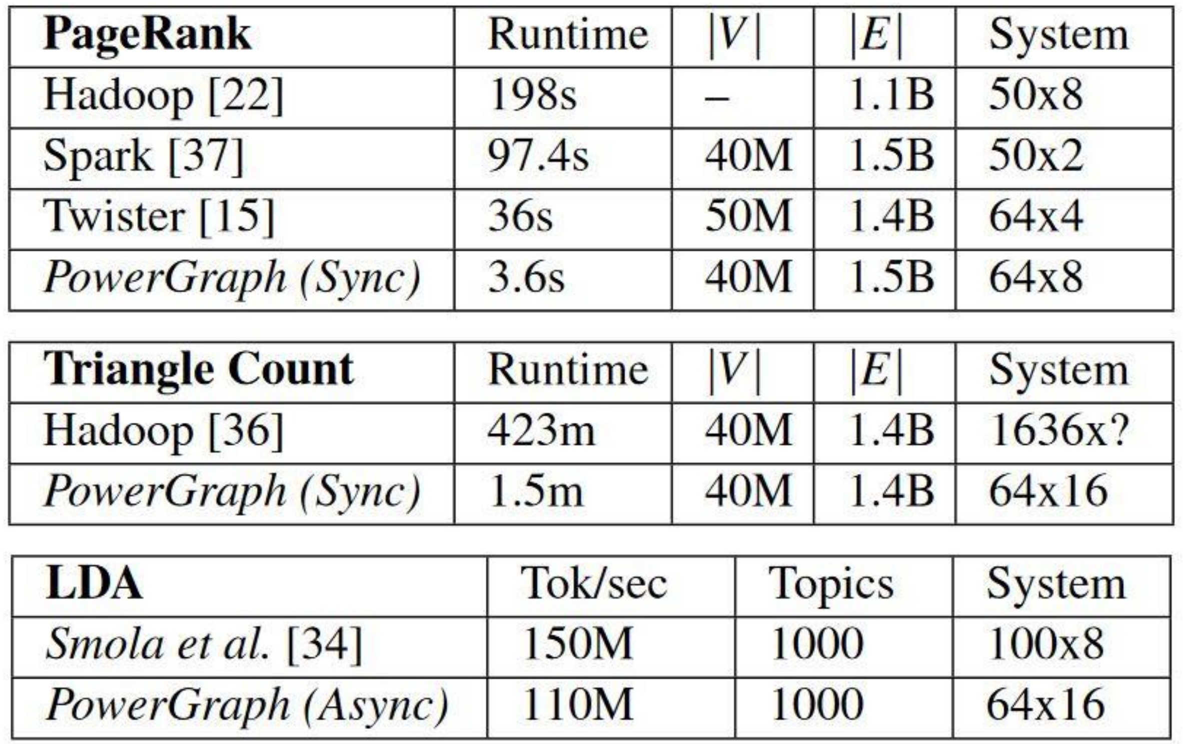
```
class pagerank :
  public graphlab::ivertex_program<graph_type, float>,
  public graphlab::IS_POD_TYPE {
  float last_change;
public:
  /* Gather the weighted rank of the adjacent page   */
  float gather(icontext_type& context, const vertex_type& vertex,
               edge_type& edge) const {
    return ((1.0 - RESET_PROB) / edge.source().num_out_edges()) *
      edge.source().data();
  }

  /* Use the total rank of adjacent pages to update this page */
  void apply(icontext_type& context, vertex_type& vertex,
             const gather_type& total) {
    const double newval = total + RESET_PROB;
    last_change = std::fabs(newval - vertex.data());
    vertex.data() = newval;
  }

  /* The scatter edges depend on whether the pagerank has converged */
  edge_dir_type scatter_edges(icontext_type& context,
                              const vertex_type& vertex) const {
    if (last_change > TOLERANCE) return graphlab::OUT_EDGES;
    else return graphlab::NO_EDGES;
  }

  /* The scatter function just signal adjacent pages */
  void scatter(icontext_type& context, const vertex_type& vertex,
               edge_type& edge) const {
    context.signal(edge.target());
  }
}; // end of factorized_pagerank update functor
```

EVALUATION



PageRank	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [22]	198s	–	1.1B	50x8
Spark [37]	97.4s	40M	1.5B	50x2
Twister [15]	36s	50M	1.4B	64x4
PowerGraph (Sync)	3.6s	40M	1.5B	64x8

Triangle Count	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [36]	423m	40M	1.4B	1636x?
PowerGraph (Sync)	1.5m	40M	1.4B	64x16

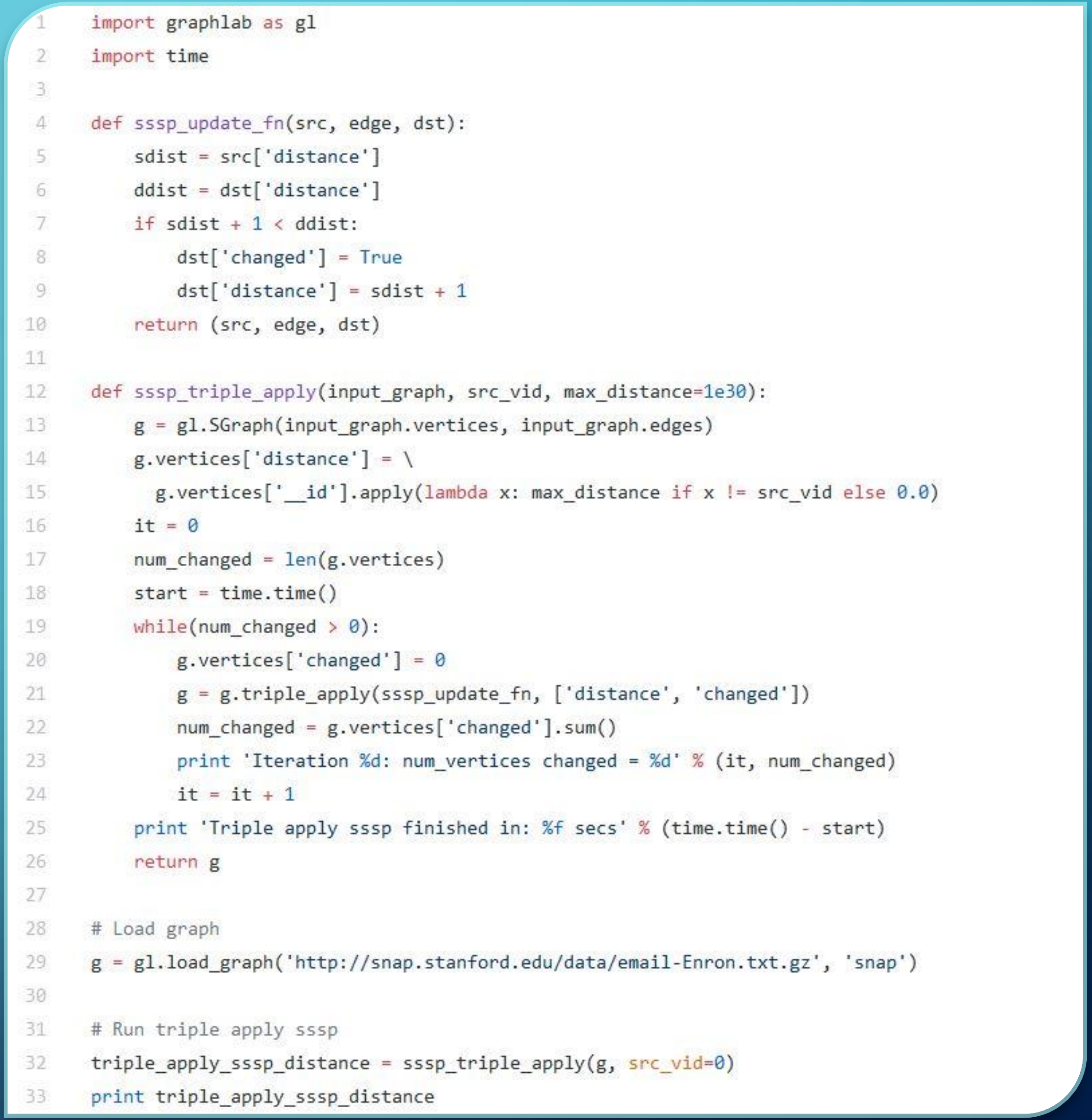
LDA	Tok/sec	Topics	System
Smola et al. [34]	150M	1000	100x8
PowerGraph (Async)	110M	1000	64x16

Table 2: Relative performance of PageRank, triangle counting, and LDA on similar graphs. PageRank runtime is measured per iteration. Both PageRank and triangle counting were run on the Twitter follower network and LDA was run on Wikipedia. The systems are reported as number of nodes by number of cores.

C++ IS DUMB, AND PAPER IS OLD

- GraphLab, inc Became Turi, a machine learning platform, now owned by Apple
- Everything is in Python Now!
- Loads data from Spark, Pandas, SQL databases, JSON, Neo4j
- More focused on machine learning now

PYTHON CODE EXAMPLE: SSSP



```python
import graphlab as gl
import time

def sssp_update_fn(src, edge, dst):
    sdist = src['distance']
    ddist = dst['distance']
    if sdist + 1 < ddist:
        dst['changed'] = True
        dst['distance'] = sdist + 1
    return (src, edge, dst)

def sssp_triple_apply(input_graph, src_vid, max_distance=1e30):
    g = gl.SGraph(input_graph.vertices, input_graph.edges)
    g.vertices['distance'] = \
      g.vertices['__id'].apply(lambda x: max_distance if x != src_vid else 0.0)
    it = 0
    num_changed = len(g.vertices)
    start = time.time()
    while(num_changed > 0):
        g.vertices['changed'] = 0
        g = g.triple_apply(sssp_update_fn, ['distance', 'changed'])
        num_changed = g.vertices['changed'].sum()
        print 'Iteration %d: num_vertices changed = %d' % (it, num_changed)
        it = it + 1
    print 'Triple apply sssp finished in: %f secs' % (time.time() - start)
    return g

# Load graph
g = gl.load_graph('http://snap.stanford.edu/data/email-Enron.txt.gz', 'snap')

# Run triple apply sssp
triple_apply_sssp_distance = sssp_triple_apply(g, src_vid=0)
print triple_apply_sssp_distance
```

WHERE TO GET IT

- Turi.com
  - Get the academic account
- https://github.com/jegonzal/PowerGraph

REFERENCES

- https://www.usenix.org/sites/default/files/conference/protected-files/gonzalez_osdi12_slides.pdf
- https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf