Chapter 1

Power Graph

Contributed by Kyle Sweeney

1.1 Background

PowerGraph came out of the limitations of Pregel and GraphLab at handling graphs which had Power-Law degree distribution, that is, graphs where most vertices have very few edges, and very few vertices have a huge number of edges. This poses a problem as traditional "graph-parallel abstractions rely on each vertex having a small neighborhood to maximize parallelism and effective partionining to minimize communication" [3]. Thus Gonzalez et. al created PowerGraph as a tool which introduces new abstractions and tools to handle these kinds of graphs. The key abstraction insight is to "[factor] computation over edges instead of vertices" [3]. The tool is a C++ library which runs as an MPI program for its multiprocessing, with users creating classes that implements their abstraction. Now days the project has been migrated and shifted focus away from graph processing to machine learning with their project "turi". Currently a fork of the old PowerGraph project can be found at https://github.com/jegonzal/PowerGraph.

1.2 Expressing Graphs

When loading graphs into PowerGraph, the default supported types are TSV, Snap, Adj, and bintsv4. TSV and SNAP are both done in similar manner. They are edge-lists where each line represents a new directed edge between two vertices. SNAP allows for comment lines whereas TSV doesn't. ADJ format is done by having each line start with the source vertex followed by each vertex which has a direct edge to it from the source vertex. The documentation claims that this is a more compact format and will "partition better in the distributed setting" [4]. It also allows vertices with no edges. Finally, the bintsv4 is a binary format composed of 8-byte blocks. The first 4 bytes, stored in x86 little endian format, is the source vertex. The second 4 bytes is the destination vertex, thus forming the edge. If a vertex has no edges, then the value of $2^32 - 1$ is used. It does support some other types, but the documentation calls these "non-portable".

One key factor in the strength of PowerGraph is how it handles graphs which obey the power law. Other libraries and paradigms split on edges, causing a high degree of shadow vertices, decreasing throughput by increasing network communication. PowerGraph splits on vertices instead of edges, splitting up vertices which have a high-degree. The idea is to have each edge only on one machine. Edge data can be stored thus only in one place, but updated vertices must be communicated. Vertex programs thus also run across multiple machines. When a vertex is replicated, a random one is the master/official version with all others being copies with read-only data. There are roughly three methods for choosing how to cut a graph based on verticies, aka, where to assign each edge.

The first method is randomly assigning edges to different nodes. The advantage to this method is that it is fully parallel in loading data, achieving "nearly perfect balance on large graphs, and can be applied to streaming." [3].

The second method is via a Coordinated Greedy Method. The algorithm keeps track of the value of $A(v)$ which represents which machines have been assigned the vertex v. The greedy heuristic follows a specific set of rules when placing edge (u, v): "**Case 1)** If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection. **Case 2)**: If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges. **Case 3)**: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex. **Case 4)**: If neither vertex has been assigned, then assign the edge to the least loaded machine." [3] In following these rules, the coordinated method will keep all the $A_i(v)$ in a distributed table. Each machine then runs the greedy method and will occasionally update the table. Caching helps speed up execution.

The Third method is via an Oblivious greedy method. It follows the same rules as above, however, there is no table maintaining $A_i(u)$.

In general, the random placement had a higher replication factor, leading to an increased runtime, while the actual construction took less time. The Coordinated greedy approach had the best runtime, but took the longest to create the graph. The figure 1.1 demonstrates these results.

1.3 Key Graph Primitives

Different types of graphs in the PowerGraph api support different things. One key graph type is the distributed graph. This graph provides for vertices and edges, allowing users to apply their own unique data to each by using a template for VertexData and EdgeData. The Documentation specifies that this data must be "Copyable, Default Constructable, Copy Constructable and sec_serializeable" [2].

For example, if the user wished to have vertices of simple double values a graph might be defined like:

```
typedef graphlab::distributed_graph<double, graphlab::empty> graphtype
```

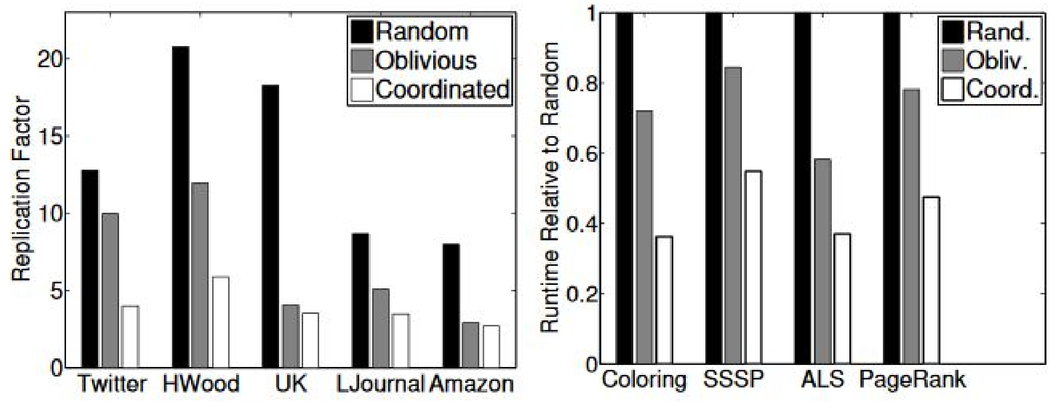
For a graph to run Sing Source Shortest Path aka Dijkstra's algorithm, the graph might look like

```
typedef graphlab::distributed_graph<String, double> graphtype
```

The other main primitive at the heart of PowerGraph's engine is the vertex program. PowerGraph operates on a "think like a vertex" paradigm, and thus the user needs to provide a vertex program to operate on the distributed graph. This vertex program should operate in the Gather-Apply-Scatter paradigm.

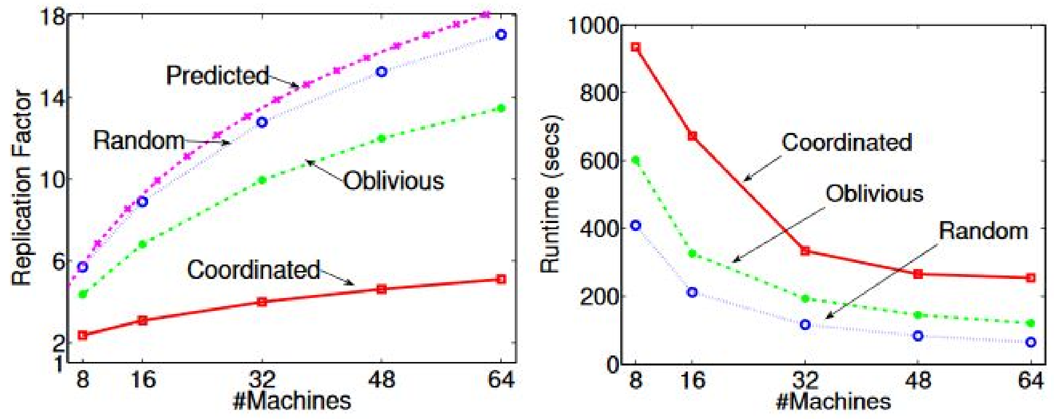
In Gather-Apply-Scatter, each vertex program is stateless and implements the "GASVertexProgram(u)". This interface has four functions: *gather(*D_u, $D_{(u,v)}$, D_v*)*, *sum(*$Accum\ left,\ Accum\ right$*)*, *apply(*$D_u$,*Accum)*, and *scatter(*D_u^{new},$D_{(u,v)}$,D_v*)*.

Gather and Sum are called during the Gather phase, acting as a Map and Reduce to "collection information about the neighborhood of the vertex" [3]. The function itself runs on the edges in parallel. The gather method returns a temporary accumulator that is summed up. That result is then cached by PowerGraph and applied in the apply phase. The new vertex value is written to the graph using an atomic write. How large a_u is, and how complicated the apply function is are determining factors in calculating how efficient PowerGraph operates at a Network and storage



(a) Actual Replication (b) Effect of Partitioning

Figure 7: **(a)** The actual replication factor on 32 machines. **(b)** The effect of partitioning on runtime.



(a) Replication Factor (Twitter) (b) Ingress time (Twitter)

Figure 8: **(a,b)** Replication factor and runtime of graph ingress for the Twitter follower network as a function of the number of machines for random, oblivious, and coordinated vertex-cuts.

Figure 1.1: Taken from the PowerGraph paper [3], the results of different replication techniques on the runtime.

Algorithm 1 Vertex-Program Execution Semantics:
Center vertex u

```
procedure BREED(CENTER U)
    if cached accumulator $a_u$ is empty then
        for neighbor v in gather_nbrs(u) do
            $a_u = sum(a_u, gather(D_u, D_{(u,v)}, D_v)$
        end
    end
    $D_u = apply(D_u, a - U)$
    for neighbor v sin scatter_nbrs(u) do
        $(D_{u,v}, \Delta a) = scatter(D_u, D_{(u,v)}, D_v)$
        if $a_v$ and $\Delta a$ are not Empty then
            $a_v = sum(a_v, \Delta a)$
        Else $a_v = Empty$
        end
    end
```

scale. Ideally it's constant in the degree of the graph, and usually sub-linear. The scatter function is then similarly applied to the edges of the vertex in parallel, updating the edge values thanks to the new vertex values. A new variable Δa is also returned to "dynamically update the cached accumulator a_v for the adjacent vertex." [3] At the scatter stage, we can reactivate our node.

The delta cache is used to help reduce the need to repeatedly call the gather operation on edges which have no need to be updated. The accumulator a_u is maintained in a cache by PowerGraph. The value Δa from the scatter phase is an optional value. If it's not returned, the neighbor's cached value of a_v is cleared. Ideally, PowerGraph tries to use the cached accumulators to avoid needing to do another gather. The paper describes Δa as "an additive correction on-top of the previous gather for that edge" [3]. Assuming that the accumulator is a part of an abelian group, then the value of Δa is $gather(D_u, D^{new}_{(u,v)}, D^{new}_v) - gather(D_u, D_{(u,v)}, D_v)$.

1.4 Execution Model

PowerGraph's underlying engine maintains a list of activated vertices on which it applies the Gather-Apply-Scatter functions. After the scatter function has been called on a vertex, the vertex is said to be deactivated. Vertices can activate themselves or any other vertex they have access to. For example, in the *scatter* function, the vertex u can activate itself and vertex v. Users can activate a specific vertex, or can activate all them with the *Activate_all()* function. The engine continues until there are no more vertices active. The authors of PowerGraph say that the underlying engine is the final decider of the exact order in which vertices will be processed, thus leading to three different execution types: Bulk Synchronous, Asynchronous, and Asynchronous Serializable.

In Synchronous mode, each step of Gather-Apply-Scatter described in section 1.3 is though of as a "minor-step" and the GAS as a whole is a "super-step" [3]. Each minor-step happens on every vertex at the same time. Barriers are imposed at the end of each minor step to ensure that every vertex is at the same place. Once the data is committed, the next minor-step can proceed. After every vertex has completed the Scatter step, then the vertices which were activated in the last super-step are then processed in the next super-step. Synchronous execution is useful for determinism, but the barriers happening after every minor-step is inefficient.

In Asynchronous mode, the GAS steps can be applied to any activated vertex if the network and workers allow it. This leads to some programs finishing much faster than in the Synchronous mode. The other version of Synchronous mode comes from GraphLab having a corresponding serial version of any parallel execution. PowerGraph simply modifies this parallel execution using parallel locking mechanisms to be fair to vertices with high-degrees.

1.5 Syntax

PowerGraph grew out of GraphLab: an MPI program which executes the individual vertex programs on all the different processes the MPi program runs on. Section 1.3 demonstrates one method for how to construct a graph .

When constructing a vertex data, the user does need to specify three things: a default constructor, and save/load functions so the vertex can become serializeable. The following example is a web page vertex type for a pagerank program, taken from the documentation from the GitHub of the project [1]:

```
struct web_page {
    std::string pagename;
    double pagerank;
    web_page():pagerank(0.0) { }
    explicit web_page(std::string name):pagename(name),pagerank(0.0){ }

    void save(graphlab::oarchive& oarc) const {
      oarc << pagename << pagerank;
    }

    void load(graphlab::iarchive& iarc) {
      iarc >> pagename >> pagerank;
    }
};
```

Given that edgedata in the documentation has the same requirements as vertex data, when using custom data types, it's a safe assumption that a similar looking struct can be used for edges.

To start up execution, the user simply creates an engine, activates the vertices they wish, and then starts. This can be seen in the example code in section 1.6.

Running the program is as simple as running an MPI Program: *mpiexec -n 10 ./PowerGraph-Program.*

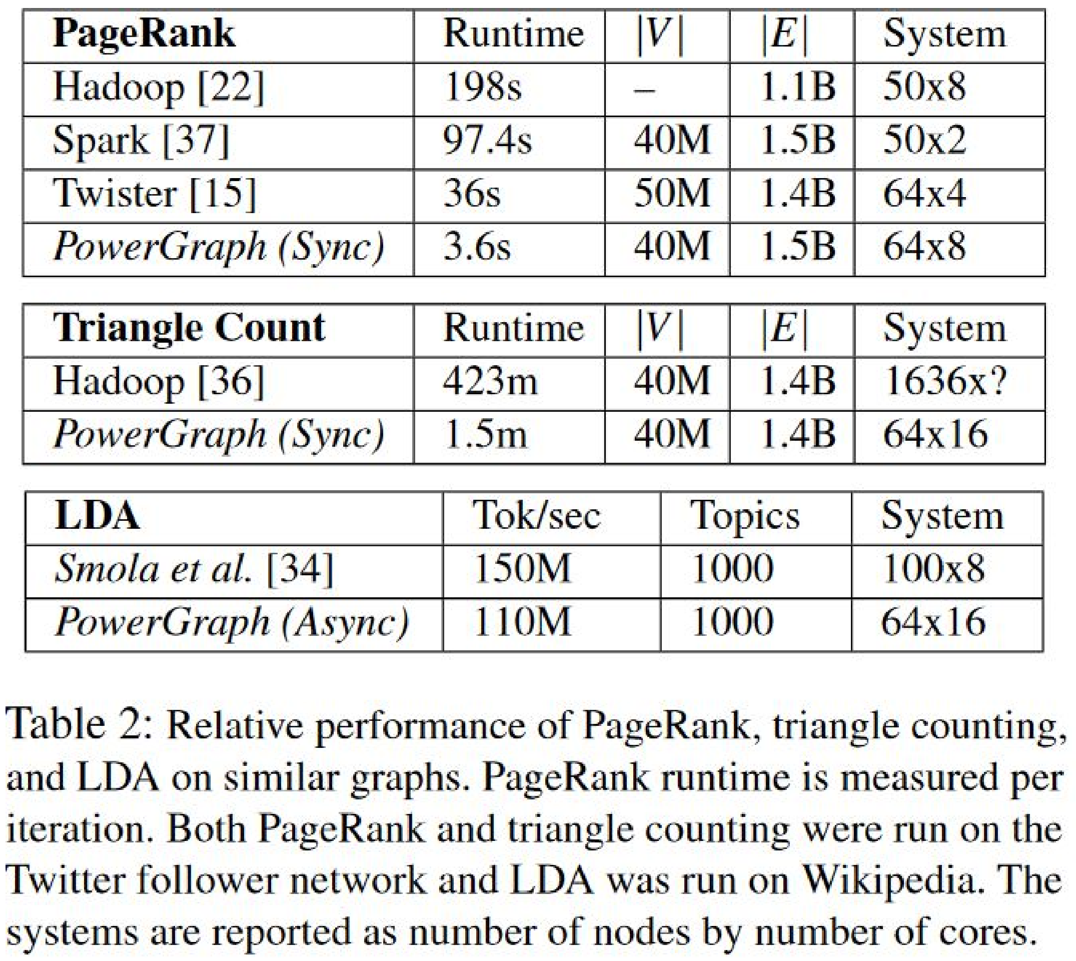
1.6 Examples

An example of how to write a page-rank program can be found on their github: https://github.com/jegonzal/Power

Figure 1.2 demonstrates the efficiency of PowerGraph over other similar systems. In general, PowerGraph can generally outperform using similar amounts of hardware, sometimes by an order of magnitude or two, such as comparing PowerGraph and Hadoop on PageRank and Triangle Count.

1.7 Conclusion

When PowerGraph was released, it was an important improvement on both GraphLab and Pregel, increasing the performance of parallel graph systems for natural data sets, such as from twitter, etc.



PageRank	Runtime	$\lvert V \rvert$	$\lvert E \rvert$	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	$\lvert V \rvert$	$\lvert E \rvert$	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.

Figure 1.2: Performance Data from PowerGraph paper [3]

Compared to other methods, such as Hadoop and Spark, it could greatly outperform them, thanks to splitting on vertices instead of edges. Their "think-like-a-vertex" almost lends to a "think-like-an-edge" paradigm, thanks to the Map/Reduce step happening on all edges of activated vertices and splitting the vertex program across multiple machines.

Bibliography

[1] Blie Arkansol. Vertex data example – from github of powergraph, 2014.

[2] Fabio Petroni. distributed_graph.hpp – from github of powergraph, 2015.

[3] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[4] Yucheng Low. Graph formats – from github of PowerGraph, 2012.