Pregel: A System for Large-Scale Graph Processing

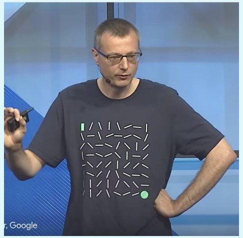
Paper Review

Presented by Alex Mcneilly

Authors (all Google)

Grzegorz Malewicz
Facebook (2012 —)
Google (2006-2012)
MapReduce, Dist. Comp.

Matthew H. Austern
Google (2005 —)
MapReduce, Distributed Computation, GCP



Aart J. C. Bik
NVIDIA (2024 —)
Google (2007-2024)
Vectorization, Compilers

Jim Dehnert
Google (2005-2016)
Exotic compilers, GCP, Performance analysis

Ilan Horn
Google (2007-2016)

Naty Leiser
Google (2006-2012)

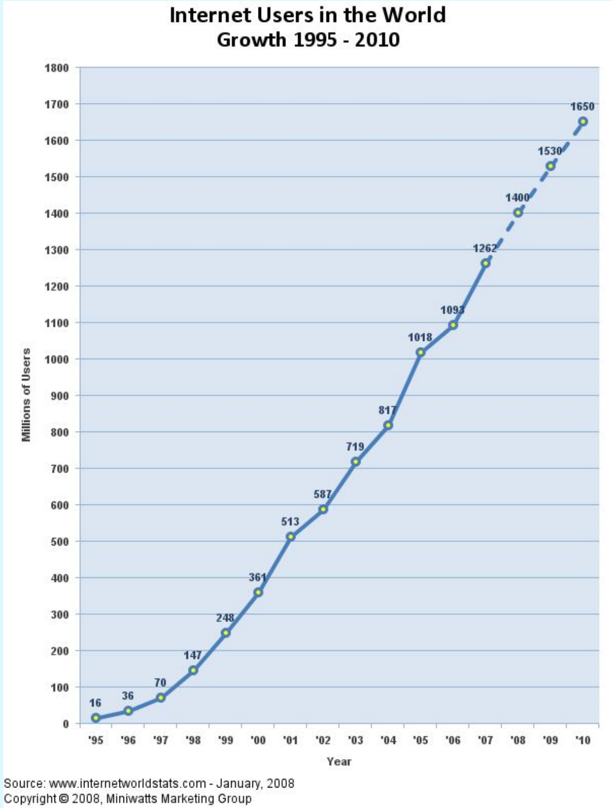
Grzegorz Czajkowski
Google (2006-2019)
GCP, BigQuery, Cluster Management

Google

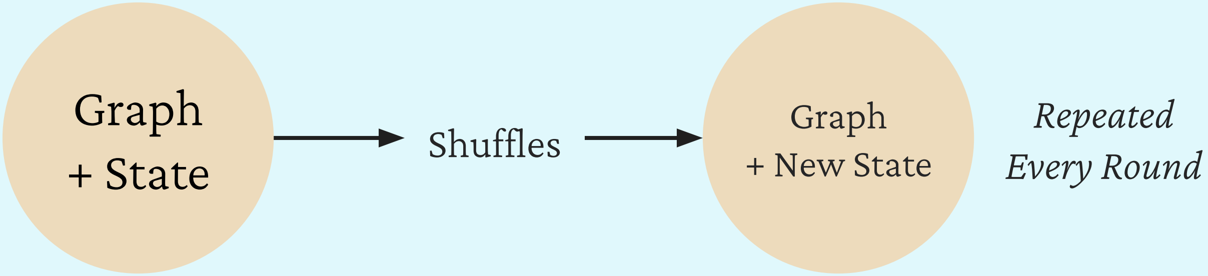
Jeff Dean
Charles Reiss
Punyashloka Biswal
Petar Maymounkov
others

Motivation

- 2008-2009: Internet grows; Google has web graphs with billions of vertices and many more edges
- Other large graphs (transportation, disease, paper citations)
- Pain points
  - Existing graph tools didn't scale
  - Poor memory locality (pointer chasing, scattering)
  - Hard to balance load for parallelism
  - Failures in clusters common
  - Iterative graph algorithms need persistent state
- Custom code (or Message Passing Interface)
- Single-machine libraries (BGL, LEDA, etc.) are elegant but limited
- Parallel graph libraries (PBGL/CGMgraph); weak fault tolerance
- Vertex-local statefulness is the missing primitive
- MapReduce?



Why MapReduce Falls Short



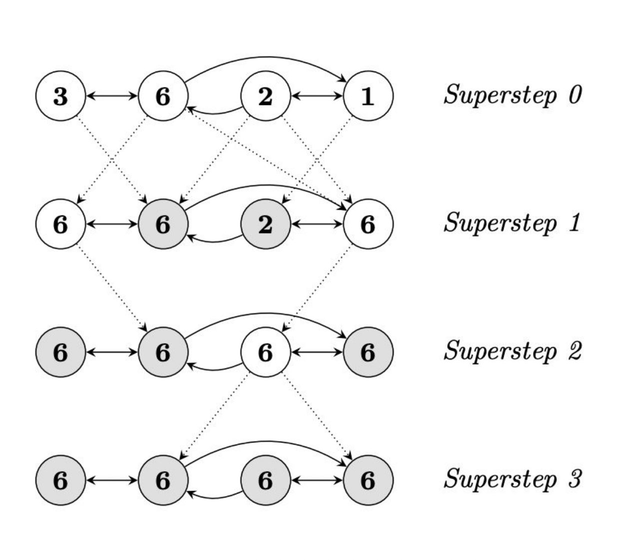
- Graph algos that are **iterative** and **stateful at vertices** vs. batch processing
- MapReduce passes the **whole graph state** across rounds
  - (might write entire graph to disk after each iteration)
- Network/serialization dominates
- Orchestration becomes cumbersome
- **Ex: PageRank**
  - Ship vertex records + adj. over and over
- Introducing: **Pregel**

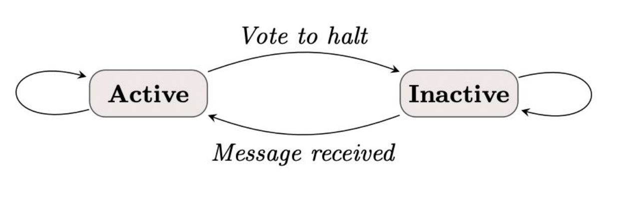
Pregel Overview

- Inspired by Bulk Synchronous Parallel (BSP)
- Superstep structure (synchronous `Compute()`)
- Messages from *S* are seen at *S+1* (no data races)
- Easier debugging

- "Think Like a Vertex"
  - Each vertex per superstep can:
    - Read last round's messages
    - Update its own state and out-edges
    - Send messages
    - Become inactive / reactivate (send/receive)
- Edges are not first-class compute units
- No remote reads

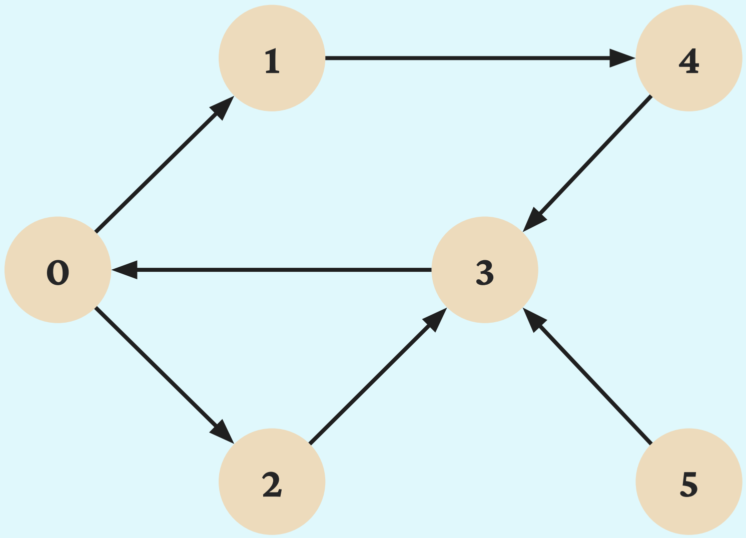
Maximum Value Example





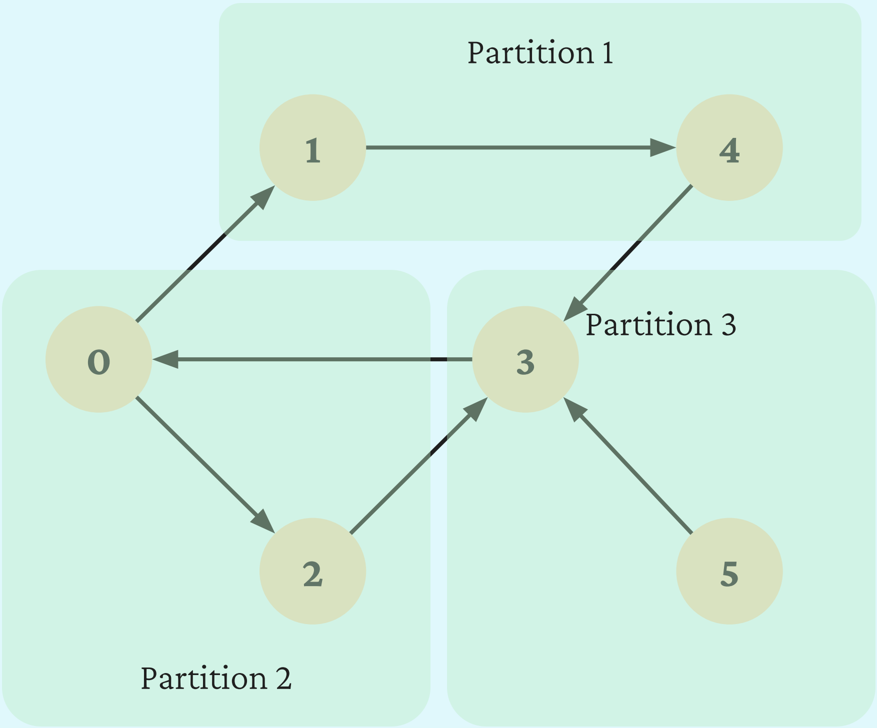
Distributed System Architecture

- **Partitioning Strategy of Input Graph**
  - Default hashing `hash(vertex_id) mod N` (random distribution)
  - Custom (domain-aware)
- **I/O Layer**
  - Pluggable readers/writers for different graph formats
- **Master:** allocates and orchestrates supersteps, and sync barriers, checkpointing + pinging (fault tolerance), confined recovery, stats server
- **Workers:** store partitions; run `Compute()` in parallel, buffer and route messages between supersteps, update state, handle signal



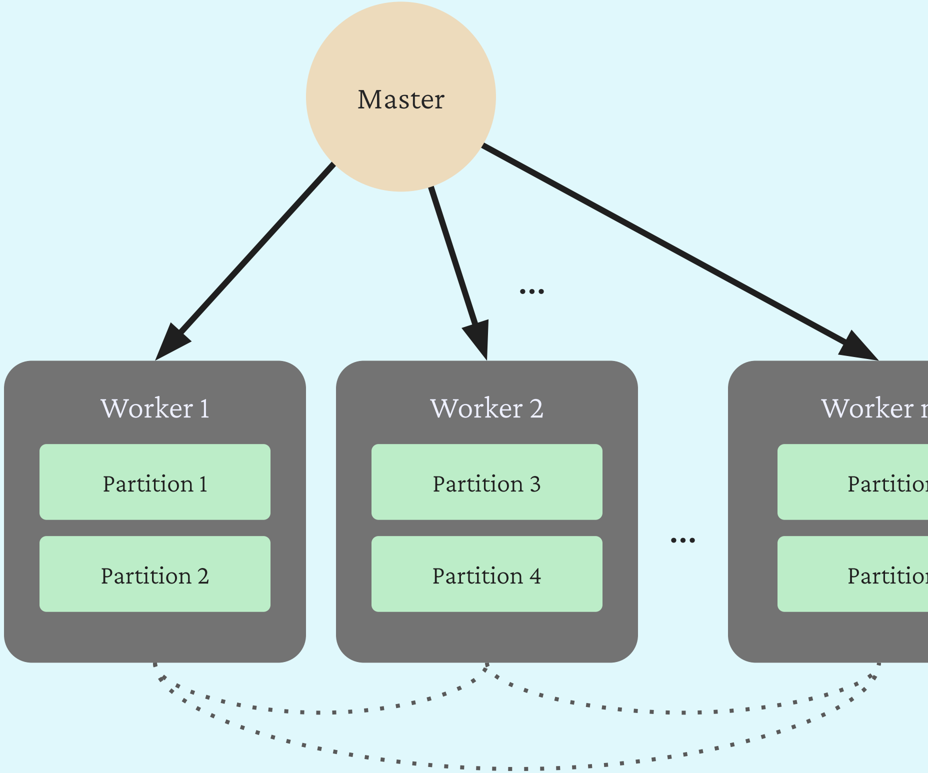
Distributed System Architecture

- **Partitioning Strategy of Input Graph**
  - Default hashing `hash(vertex_id) mod N` (random distribution)
  - Custom (domain-aware)
- **I/O Layer**
  - Pluggable readers/writers for different graph formats
- **Master:** allocates and orchestrates supersteps, and sync barriers, checkpointing + pinging (fault tolerance), confined recovery, stats server
- **Workers:** store partitions; run `Compute()` in parallel, buffer and route messages between supersteps, update state, handle signal



Distributed System Architecture

- **Partitioning Strategy of Input Graph**
  - Default hashing `hash(vertex_id) mod N` (random distribution)
  - Custom (domain-aware)
- **I/O Layer**
  - Pluggable readers/writers for different graph formats
- **Master:** allocates and orchestrates supersteps, and sync barriers, checkpointing + pinging (fault tolerance), confined recovery, stats server
- **Workers:** store partitions; run `Compute()` in parallel, buffer and route messages between supersteps, update state, handle signal



Core API (C++)

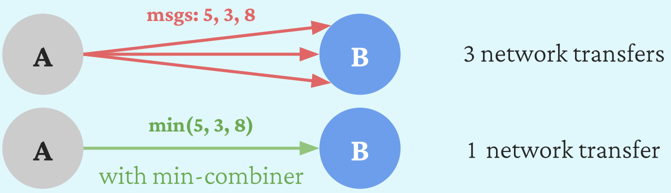
- **Combiners:** Combine messages; pre-aggregate messages per destination; must be associative + commutative
- **Aggregators:** Good for global values; Master collects all values, reduces them, and broadcasts the result between $S \longrightarrow S+1$
- **Topology Mutations:** Add/Remove vertices or edges, contract clusters
  - Applied deterministically in a fixed order between supersteps to prevent conflicts

```
class Vertex<V, E, M> {
    void Compute(MessageIterator* msgs);
    void SendMessageTo(Id dest, const M& m);
    void VoteToHalt();

    const V& GetValue();
    V* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();
    int64 superstep();
}
```

Core API (C++)

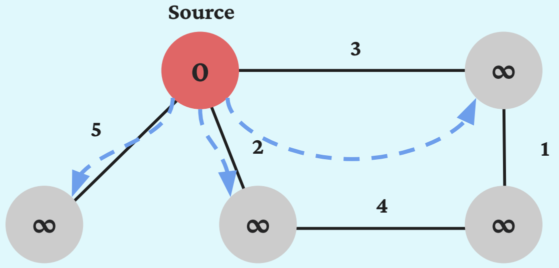
- **Combiners:** Combine messages; pre-aggregate messages per destination; must be associative + commutative **(4x reduction in msg traffic)**
- **Aggregators:** Good for global values; Master collects all values, reduces them, and broadcasts the result between $S \longrightarrow S{+}1$
- **Topology Mutations:** Add/Remove vertices or edges, contract clusters
  - Applied deterministically in a fixed order between supersteps to prevent conflicts



Example: Single-Source Shortest Paths (SSSP)

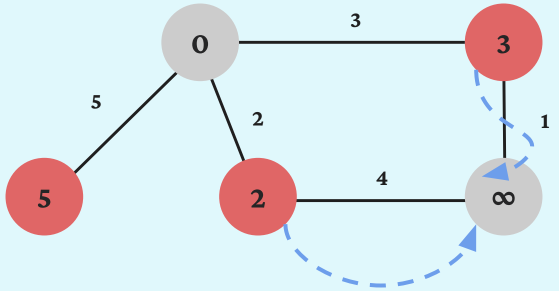
Superstep 0:

- The source vertex is the only active vertex
- Sends its distance + edge weight to all neighbors



Superstep 1:

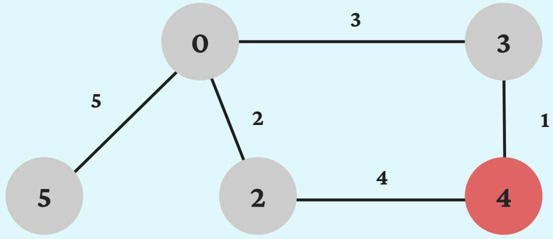
- Three vertices receive updates and become active
- Compare received values to current and update
- Active vertices send their dist + edge weights to neighbors
- Min-combiner would merge these into single message: `min(4, 6) = 4`



Example: Single-Source Shortest Paths (SSSP)

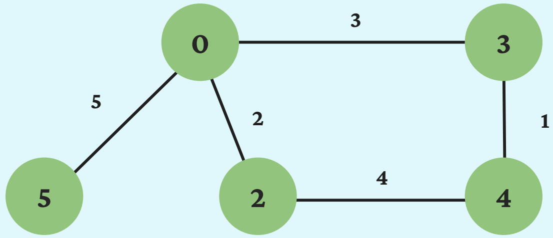
Superstep 2:

- Bottom-right vertex receives message(s) and updates to 4
- No further improvements possible
- No vertices send messages (all vote to halt)



Superstep 3:

- All vertices are inactive (green = final state)
- Algorithm terminates, shortest paths found



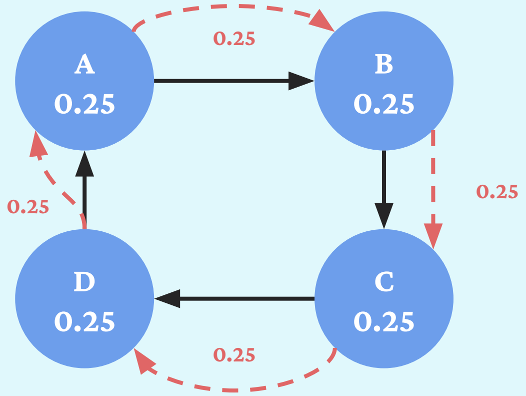
Example: Single-Source Shortest Paths (SSSP) Code

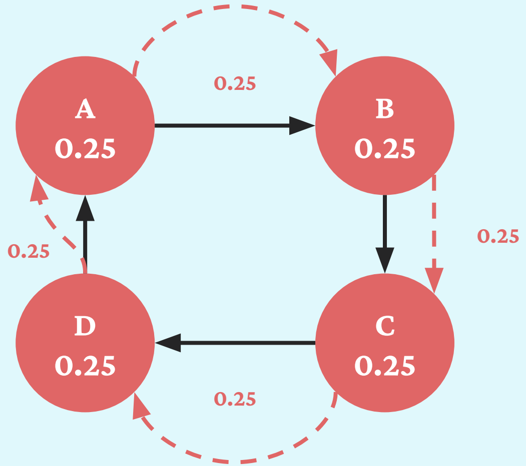
```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

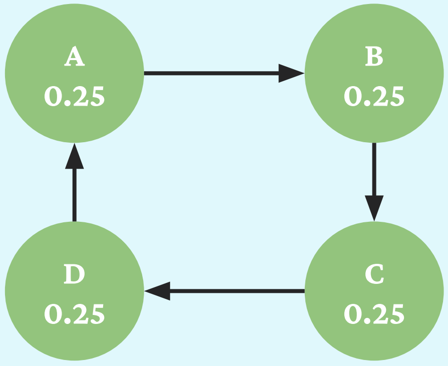
```
class MinIntCombiner : public Combiner<int> {
  virtual void Combine(MessageIterator* msgs) {
    int mindist = INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    Output("combined_source", mindist);
  }
};
```

Example: PageRank

- Models web importance through iterative voting
- **Superstep 0:** initialize all vertices to 1/N = 0.25. Each vertex sends rank/outdegree = 0.25/1 = 0.25; Perfect cycle, uniform rank
- **Superstep 1:** Apply the PageRank formula; No change because of symmetry;
- Continuous until convergence check
- Network efficient (only rank contributions move)
- Graph structure stays in memory
- *Maximal Bipartite Matching and Semi-Clustering problems discussed in-depth in the paper*







Alexander Mcneilly

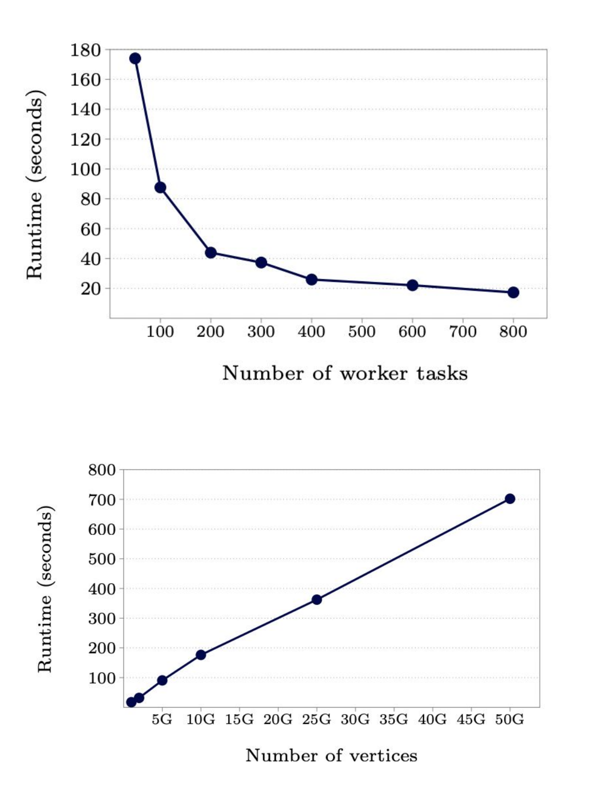
Example: PageRank Code

```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Experimental Results

- **1B-Node Binary Trees**
  - 300 multi-core machines
  - Scaling with workers (varies from 50 to 800)
  - Drop from 174 seconds to 17.3 seconds using 16 times as many workers (~10x speedup)
- **Binary Trees**
  - 300 multi-core machines
  - Scaling graph size (1B to 50B nodes)
  - Fixed number of 800 worker tasks
  - Increase from 17.3 to 702 seconds
  - Graphs with low average outdegree have a runtime that increases linearly with the graph size

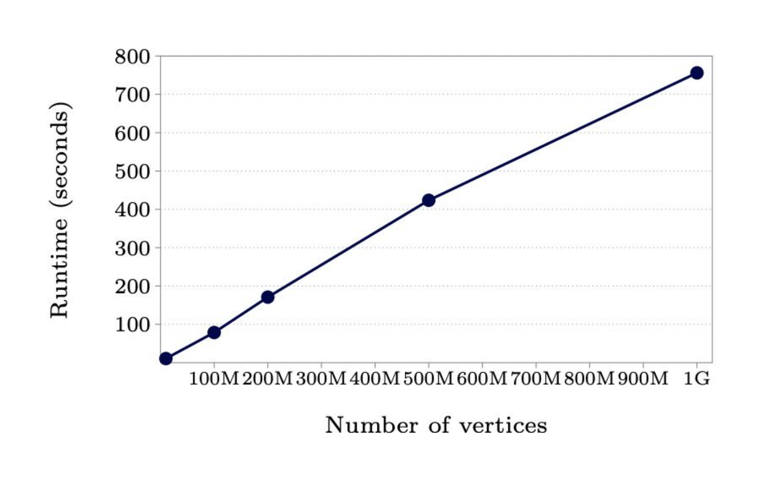


Experimental Results

- **Random Graphs**
  - Used log-normal distribution of outdegrees

$$p(d) = \frac{1}{\sqrt{2\pi}\,\sigma d}\, e^{-(\ln d - \mu)^2 / 2\sigma^2}$$

- The distribution meant to resemble real-world large-scale (web) graphs.
- Shortest paths runtimes varying in size from 10M to 1B nodes
- 800 workers
- 300 multi-core machines
- Largest graph took over 10 minutes.



Related Work

- **MapReduce / Pig Latin / Sawzall / Dryad**
  - Approachable, but stateless
  - Not good for iterative graphs
- **Other BSP Libraries**
  - More general model rather than vertex-centric
  - Doesn’t handle super large graphs
  - Not graph-specific or adaptable
- **Parallel BGL, CGMgraph**
  - Powerful, but MPI-heavy (cumbersome)
  - Exposes distribution rather than hides it
  - Fault tolerance is questionable

Strengths + Weaknesses

- **Strengths**
  - “Think like a vertex” is radically simple
  - Vertex-centric code is much faster for iterating and testing
  - Fault tolerance, monitoring, aggregators included with the framework
  - Scalability
- **Weaknesses**
  - Synchronous barriers (stragglers block progress)
  - Default partitioning could ignore locality due to rand.
  - Evaluation seems weak***
  - Not open source (then)

Future Directions + Questions

- **Future Directions**
  - Partial/async execution to relax barriers (A 2015 Waterloo paper found a 5-10x speedup over previous systems)
  - Topology-aware partitioning (paper recommends this)
  - Applications for vertex-centric machine learning or linear algebra (like GraphBLAS)
- **Discussion**
  - What kind of algorithms or workloads are naturally vertex-local?
  - Which ones *aren't* a good fit?
  - If one worker is slow, what could be adjusted (among partitioning, checkpoint interval, batching) to reduce barrier stalls?
  - How does the default partitioning strategy compare to grouping by topology?