Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

Reviewed by Miranda Cai
Large Graphs are Everywhere

We can use graphs to represent

- Social networks
- Transportation routes
- Citation maps between published work
- Disease outbreaks

with billions of vertices and edges each. But graph algorithms do a poor job with

- Memory access locality
- Optimizing parallel allocation
- Distribution over multiple machines
Pregel

Idea: A framework that executes the same user-defined function `Compute()` for each vertex in a sequence of **supersteps** until the algorithm reaches completion.

A **superstep** is a synchronous iteration that performs `Compute()` on all **active** vertices at the step.

Pregel utilizes **message passing** to communicate updates in the state of the graph immediately.
Pregel: Maximum Value Example

All vertices begin as **active**
At superstep $S$, vertices send messages to outgoing edges at superstep $S+1$. 

Superstep 0: 
- Vertex 3 sends message 6 to vertex 2. 
- Vertex 6 sends message 6 to vertex 1.

Superstep 1: 
- Vertex 6 sends message 6 to vertex 1.
Pregel: Maximum Value Example

At superstep $S$, vertices send messages to outgoing edges at superstep $S+1$.

Vertices that do not change are *voted to a halt*.
Pregel: Maximum Value Example

Superstep 0

Superstep 1
Pregel: Maximum Value Example

A vertex can become active again if another vertex sends it a message at superstep $S-1$. 

Superstep 0

Superstep 1

Superstep 2
Pregel: Maximum Value Example

The process completes once all vertices are deactivated simultaneously.
Pregel Framework: API Details

- Message passing between vertices
  - Messages sent to $V$ at $S$ are iterated through at $S+1$
  - Non-neighbors can send messages
- Topology Mutations
  - Edge removals, vertex removals, vertex addition, edge addition
  - Partial ordering and handlers to avoid data races
- `Combine()` to condense several messages into one
- `Aggregator()` for global coordination
- Support for flexible input/output graph formats
Pregel Architecture: A Distributed System

The input graph is broken into partitions, where each vertex is assigned a partition based on the hash value of its vertex ID.
Program Execution

At the start, one worker machine is assigned master. A master must

- Divide and allocate partitions to the workers
- Instruct each worker to perform a superstep
- Instruct workers to save its state

At each superstep, each worker is in charge of

- Maintaining the state of its own partition
- Sending messages to remote peers
- Loop through its active vertices and call `Compute()`
- Signal to the master when complete
Fault Tolerance

Basic **Checkpointing**

- Failed worker at $S'$: Master $\rightarrow X \rightarrow$ Worker $- - - -$ Master
- Recover supersteps since most recent checkpoint $S$

**Confined Recovery**

- Workers log their outgoing messages
- Recover from $S$ to $S'$ only for the lost partitions
- Adds overhead --
- Saves compute resources ++
Applications to Real Problems
PageRank

**Problem**: Ranking webpages based on the quality of quantity of links to the page

**Vertex**: Potential page rank, all initialized the same

**Outgoing messages**: Inversely proportional to the number of outgoing edges
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();
        }
    }
};
Single-Source Shortest Paths (SSSP)

Problem: Finding the shortest distance between a source and all other vertices

Vertex: distance from source initialized to INF

Outgoing messages: Potential minimum distances + its own edge weight

Uses a Combiner() to reduce data sent
Single-Source Shortest Paths (SSSP)

```cpp
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();
    }
};
```

```cpp
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);
    }
};
```

*Much better than single-machine implementations*
Maximal Bipartite Matching

**Problem**: Find a set of edges such that no two edges share an endpoint from a bipartite graph, using the maximum number of edges

**Vertex**: \((<L/R>, <\text{matched_vertex_ID}>)\)

**Outgoing messages**: boolean

Supersteps work in cycles of 4 phases
Maximal Bipartite Matching

L

R
Maximal Bipartite Matching

Phase 0 (Superstep 0)
Maximal Bipartite Matching

Phase 1 (Superstep 1)
Maximal Bipartite Matching

Phase 2 (Superstep 2)
Maximal Bipartite Matching

Phase 3 (Superstep 3)
Maximal Bipartite Matching

Phase 0 (Superstep 4)
Semi-Clustering

**Problem**: Finding groups of people who interact frequently with each other and less frequently with others

**Vertex**: list of at most $C_{\text{max}}$ semi-clusters sorted by score

**Outgoing message**: its semi-cluster $c$

**Score**: $S_c = \frac{I_c - f_B B_c}{V_c(V_c - 1)/2}$

- $I_c = \text{sum of weights of edges within } c$
- $B_c = \text{sum of weights of edges outgoing } c$
- $V_c = \text{number of vertices in } c$
- $f_B = \text{boundary edge score factor}$
  (parameter between 0-1)
Semi-Clustering

- Vertex $V$ iterates over the semi-clusters $c_1, \ldots, c_k$ sent to it on the previous superstep. If a semi-cluster $c$ does not already contain $V$, and $V_c < M_{\text{max}}$, then $V$ is added to $c$ to form $c'$.

- The semi-clusters $c_1, \ldots, c_k, c'_1, \ldots, c'_k$ are sorted by their scores, and the best ones are sent to $V$’s neighbors.

- Vertex $V$ updates its list of semi-clusters with the semi-clusters from $c_1, \ldots, c_k, c'_1, \ldots, c'_k$ that contain $V$. 

Experiments

Evaluated performance for SSSP on binary trees using clusters of 300 multicore machines
Experiments

Evaluated performance for SSSP on log-normal distribution of outdegrees to better represent real world graphs

\[ p(d) = \frac{1}{\sqrt{2\pi} \sigma d} e^{-\frac{(\ln d - \mu)^2}{2\sigma^2}} \]
Comparison to Existing Models

MapReduce

-- No graph API

Bulk Synchronous Parallel (BSP)

++ Same synchronous superstep model

-- No graph-specific API

-- Not tested beyond dozens of machines
Comparison to Existing Models

Parallel Boost Graph Library (BGL)

++ Implements multiple algorithms on MPI
-- Uses *ghost cells*, can cause scaling issues
-- Poor fault tolerance

CGMgraph

++ Implements multiple algorithms on MPI
-- Not generic, user cannot implement their own algorithms
Conclusion

Strengths:
- Flexible and intuitive API explanation
- Simple applications to real problems

Weaknesses:
- Shallow evaluations
- Message passing catered towards sparse graphs only

Future Directions:
- Scaling to even larger graphs
- Topology-aware partitioning