PREGEL: A SYSTEM FOR LARGE-SCALE GRAPH PROCESSING

G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski
Google, Inc.
SIGMOD 2010

Presented by Ke Hong
(some figures borrowed from Peixiang Zhao, UIUC)
OUTLINE

- Problem
- Motivation
- Background
  - MapReduce Framework
- Design
- Evaluation
  - PageRank Algorithm
  - Single-Source Shortest Path Algorithm
- Summary & Discussion
**Problem**

- To process large-scale vertex-centric web graphs
  - Sparse graph
  - Billions of vertices
  - Trillions of edges
  - Terabyte-level in-memory processing
  - PageRank, single-source shortest paths, bipartite matching, clustering
  - Very little work per vertex
OBJECTIVE

Scalable
  • Involve many machines to process in parallel
  • Together process a large-scale graph

Fault Tolerance
  • What to do when some machines are down
  • How to recover failed subtasks with minimum effort
STATE OF THE ART

- Craft a custom distributed infrastructure
  - Substantial implementation effort
  - Hard to extend or reuse

- Use existing distributed computing platform
  - MapReduce
  - Ill-suited for large-scale graph processing
**MapReduce**

- Distributed computing framework
  - Process large-volumes of data in key-value pairs
  - Simple dependency among data

- Two-phase processing
  - Map
  - Reduce
Map Phase

Partition data to each worker based on its keys
Apply a same operation to each partition

$$\text{map (lambda (x) (* x x)) (1 2 3 4 5)} \rightarrow (1 4 9 16 25)$$
**Reduce**

- **Reduce Phase**
  - Map output sent to specified reducer based on its keys
  - Apply some aggregate operation to map output
  - Return single-valued output

\[
\text{(reduce } + 0 \ '((1 \ 2 \ 3 \ 4 \ 5))) \rightarrow 15 \\
\text{(reduce } * 1 \ '((1 \ 2 \ 3 \ 4 \ 5))) \rightarrow 120
\]
MApREDUCE

$1^2 + 2^2 + 3^2 + 4^2 + 5^2 = ?$

$m: x^2$

$r: +$

Initial value 0 1 5 14 30 55

final value
**MapReduce Workflow**

- **Single Master node**
- **Worker threads**

- **Input files**
- **Map phase**
- **Intermediate files (on local disks)**
- **Reduce phase**
- **Output files**

Diagram shows the process flow of MapReduce, including the interaction between the master node and worker threads.
MAPREDUCE EXAMPLE

Massive parallel processing made simple

- Example: world count
- Map: parse a document and generate <word, 1> pairs
- Reduce: receive all pairs for a specific word, and count

Map

// D is a document
for each word w in D
output <w, 1>

Reduce

Reduce for key w:
count = 0
for each input item
  count = count + 1
output <w, count>
Graph Processing by MapReduce

- Graph algorithms written as a series of chained MapReduce iterations
  - After each iteration, need to pass entire states of the graph to the next iteration
  - Need to coordinate between steps of a chained MapReduce (i.e. scheduling, backup)
  - Communication and serialization overhead, suboptimal performance
  - Unnatural, add programming complexity
STATE OF THE ART

- Use a single-computer graph algorithm library
  - BGL, LEDA, NetworkX, JDSL
  - Not scalable

- Use existing parallel graph system
  - Parallel BGL, CGMgraph
  - Not fault tolerant
**Pregel Model**

- **Bulk Synchronous Parallel**
  - Series of synchronous iterations (supersteps)
  - Vertex asynchronously executes some user-defined function in parallel in each superstep

- **Message-passing Model**
  - Vertex reads messages sent in previous superstep
  - Vertex sends messages, to be read by other vertices in the next superstep
  - Vertex updates states of itself and its outgoing edges
**Vertex State Machine**

- Execution stops when all vertices have voted to halt and no vertices have messages.
**MESSAGE PASSING**

- Simple example: finding maximum vertex

![Graph with supersteps and code snippets](image)
**Compute Model**

- **Pregel Input & Output**
  - A directed graph as input
  - Processed directed graph as output

- **Worker & Master**
  - Master assigns a portion of the graph to each worker
  - Worker processes its assigned portion of graph in memory
Vertex Partition
**MASTER**

- Partition the graph and assign input to workers
- Keep track of which worker holds which portion
- Initiate each superstep
- Maintain task statistics and run an HTTP server for users to view job information
Worker

- Load its portion of graph into memory
- Receive messages from neighboring vertices
- Update states of vertices, edges
- Queue in messages for next superstep
**COMBINER**

- Worker can combine messages reported by its vertices and send out one single message
  - Reduce message traffic and network bandwidth overhead
AGGREGATOR

- Upper-level workers can aggregate output from lower-level workers
  - Further reduce message traffic and bandwidth overhead
Fault Tolerance

- Checkpoint at each superstep
  - Worker saves states of its vertices, edges, incoming messages into persistent storage
  - Master saves aggregator values (if any)
  - Costly to do at every superstep (checkpoint frequency using mean time of failure)

- When master detects worker failures
  - All workers revert to last checkpoint and continue from there
  - Might involve a lot of repeated work
Fault Tolerance

Confined Recovery

- Workers also log outgoing messages at each superstep and recover only the lost partitions
- Other workers re-send messages sent to failed worker at each superstep occurring after the last checkpoint
- Failed worker catch up to the rest, but still have to wait on failed workers to catch up
- Less use of resources
- Reduce recovery overhead and latency
PageRank in Pregel

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();
        }
    }
};

\[
PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)}
\]

http://wikipedia.org
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();
    }
};
**Shortest Path in Pregel**

At each superstep...

![Diagram](image)

(vertex receives messages)

if \(\min(d_0, d_1) < d_v\), it sends messages to its neighbors and updates its new minimum distance from \(s\)

else, it votes to halt

After execution, each vertex's value is its minimum distance from \(s\)
SUMMARY

- Pregel as a message passing model to efficiently process large-scale graphs
- Pregel as a scalable master-worker model for large-scale graph problems
- Pregel as a fault-tolerant distributed graph processing framework


**Discussion**

- Pregel currently loads and processes the entire graph states in memory

- Pregel currently focuses on sparse graphs
  - What about dense graphs?
  - How to handle heavy all-to-all message traffic?

- Pregel has not partitioned graph based on topology
  - What if the topology does not correspond to the message traffic?