Motivation

Many applications require graph processing
E.g., PageRank

Some graph data sets are very large
E.g., the Web

→ Need distributed graph processing

MapReduce is not a good fit for graph processing

Pregel

_Pregel: A System for Large-Scale Graph Processing._ G. Malewicz et al., SIGMOD 2010

Open-source version: Apache Giraph
Pregel

Distributed computational model/infrastructure for processing large graphs

E.g., PageRank on the web graph

\[ pr(x) \] = \[ \frac{pr(a)}{\text{out}(a)} + \frac{pr(b)}{\text{out}(b)} \]

Pregel

Synchronous computation based in iterations

In one iteration, each node
1. Gets messages from neighbors
2. Performs computation (updates its state)
3. Sends update messages to neighbors

Execution Flow

In MapReduce and Spark the execution flow is not a function of the data itself

In Pregel the data (graph structure) drives the execution flow

Execution Flow

Example

Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.
Termination

After each iteration, each vertex votes to halt or not

If all vertexes vote to halt, computation terminates

Vote to halt

Active

Inactive

Message received

Computation at a Node

Data available in iteration i:

- `outEdges`: `[[vertex, edgeWeight]]`
- `value`: ...
- `inputMessages`: `[[fromVertex, inputValue]]`
- `outputMessages`: `[[toVertex, outputValue]]`

OutEdges and value are remembered for next iteration

Example: PageRank

```java
class PageRankVertex
  extends Vertex<
    double, void, double> {
  public:
    virtual void Compute(MessageIterator msgs) {
      if (superstep() >= 1) {
        double sum = 0;
        for (msg msg : msgs) {
          msg += msg.next();
          sum += msg.value();
          mValue() =
            0.15 / numVertices() + 0.85 * sum;
        }
        if (superstep() < 20) {
          for (int i = 0; i < msg.size(); i++)
            msg.setValue(msg / n);
        } else {
          VoteToHalt();
        }
    }
```
Example: PageRank

class PageRankVertex:
  public Vertex<double, void, double> {
    public:
      virtual void Compute(MessageIterator* msgs) {
        if (superstep() > 0) {
          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();
        }
      }
    }
}

Example: Single-Source Shortest Path

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();
    }
  }

Example: Single-Source Shortest Path

Example: Single-Source Shortest Path

Architecture

Graph has nodes a, b, c, d...

Sample record: [a, value, edges]
Partition graph and assign to workers

Read input data

Run superstep 1

Worker 1 forwards input values to appropriate workers
Worker 1 sends messages at the end of superstep 1.

Worker 1

Vertices: a, b, c

Worker 2

Vertices: d, e

Worker 3

Vertices: f, g, h

Worker 1

Vertices: a, b, c

Worker 2

Vertices: d, e

Worker 3

Vertices: f, g, h

Halt?

Master

Checkpoint

Log

Each worker writes to stable storage its state, edges (in or out), and messages (incoming or outgoing).
Architecture

Master
If worker dies, find replacement and restart from latest checkpoint

Worker 1
Vertices: a, b, c

Worker 2
Vertices: d, e, f

Worker 3
Vertices: g, h

Distribution Challenge

How to best to partition graph for efficient processing?

Limitations

Graph processing often part of larger flow
Often need to join with unstructured/tabular data
Spanning multiple systems
Less opportunity for optimization
Extra data movement
No granular fault tolerance
Snapshot recovery only

GraphX

Graph processing on top of Spark

*GraphX: Graph Processing in a Distributed Dataflow Framework.* J.E. Gonzalez et al., OSDI 2014
GraphX

Graph representation
Vertex collection \{ \{ vertex, value \} \}
Edge collection \{ \{ fromVertex, toVertex, weight \} \}

Graph computation
Data flow of join and group by (punctuated by map operations)
1. Join vertex and edge collections to form triplets
   \{ fromVertex, toVertex \} → \{ fromValue, toValue, weight \}
2. Group triplets by source or destination vertex

GraphX

Optimizations
Horizontal partitioning of collections by applying vertex cut
   Assign edges to machines that already have the vertices
   If multiple machines have the same vertex, pick less loaded
   Performs better than random edge partitioning of natural graphs
Efficient joins
Efficient materialized view management
   E.g., delta update propagation
Lineage for fault tolerance

Summary

Motivation
Pregel
   Computation model
Examples
   Architecture
GraphX