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

presented by Cong Guo
March 3, 2015
Outline

- Motivation
- Model of Computation
- The C++ API
- Implementation
- Applications
- Experiments
- Conclusion
- Discussion
Google needs applications that perform Internet-related graph algorithms

Processing a large graph is challenging
  - Poor locality of memory access
  - Very little work per vertex
  - A changing degree of parallelism over the course of execution
Motivation

Four options (at 2010)

- Writing a custom infrastructure
- Using a distributed computing platform like MapReduce
- Using a single-computer graph algorithm library
- Using an existing parallel graph system
Bulk Synchronous Parallel model
- A computation proceeds in a series of global supersteps
- Three components: local computation, communication, barrier synchronization
Model of Computation

Model of Computation

- Take a graph as input
- Run at each vertex in parallel (Think as vertex)
- Run until vertices vote to halt
- Finish with output

![Graph Processing Diagram]

Vote to halt

Message received

Pregel: A System for Large-Scale Graph Processing
Within each superstep, a vertex can

- Modify its state or that of its outgoing edges
- Read messages sent to it in the previous superstep
- Send messages to other (to be received in the next superstep)
- Mutate the topology of the graph
Model of Computation

Superstep 0

Superstep 1

Superstep 2

Superstep 3
The C++ API

- **Vertex class**

```cpp
template <typename VertexValue,
    typename EdgeValue,
    typename MessageValue>

class Vertex {
    public:
        virtual void Compute(MessageIterator* msgs) = 0;

        const string& vertex_id() const;
        int64 superstep() const;

        const VertexValue& GetValue();
        VertexValue* MutableValue();
        OutEdgeIterator GetOutEdgeIterator();

        void SendMessageTo(const string& dest_vertex, const MessageValue& message);
        void VoteToHalt();
};
```
The C++ API

- **Message Passing**
  - No guaranteed order of messages in the iterator
  - Guarantee that messages will be delivered once
  - Can send messages to any vertex
  - User handlers are executed for the missing vertex

- **Combiners**
  - Not enabled by default
  - Combine multiple messages to the same vertex into a single one
  - Only for commutative and associative operations
Aggregators

- A mechanism for global communication, monitoring, and data
- Each vertex sends a value to an aggregator in superstep $S$
- All vertices receive the resulting value in superstep $S + 1$
- Can be used for statistics and global coordination
The C++ API

- **Topology Mutations**
  - Vertices can issue requests to add or remove vertices or edges
  - Resolving conflicting requests in the same superstep:
    - Partial ordering - edge removal before vertex removal; vertex addition before edge addition
    - User-defined handlers
  - Local mutations have no conflicts

- **Input and output**
  - Support various file formats, even custom Reader and Writer
**Basic architecture**

- Copies of user program are sent to the cluster - master/workers
- Master assigns graph partitions to workers - vertex partition
- Master assigns a portion of user input to each worker
- Supersteps begin
- Save the output graphs
Fault tolerance

- Achieved through checkpointing
- At the beginning of a superstep:
  - Workers persist the state of their partitions
  - Master saves the aggregator values
- If one or more workers fail, everyone starts over from the most recent checkpoint
- Confined recovery with message logs is under development
The Worker

- A worker keeps its portion of the graph in memory
- Two copies of the active vertex flags and the incoming message queue for the current and next superstep
- Messages to a remote worker are buffered
- Combiner may be used
The Master

- Keeps track of which portion of the graph a worker is assigned
- Coordinates the activities of workers using barriers
- Maintains the statistics of the progress and the graph for user monitoring
Aggregators

- An aggregator computes a global value with values from workers
- Workers form a tree to reduce partially reduced aggregators
- A tree structure is better than chain pipelining
Applications - PageRank

- Ranks web pages according to their popularity
- Named after Larry Page instead of Web Page
- Computes the page rank of every vertex in a directed graph iteratively
- At every iteration, each vertex computes its rank according to its neighbors’ rank values at last iteration
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();
        }
    }
};
Applications - Single Source Shortest Paths

- Parallel breadth first search

```java
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();
        }
    }
};
```
Applications - Bipartite Matching

- Input: a bipartite graph with two distinct sets of vertices
- Output: a subset of edges with no common endpoints
- Vertices maintain two values: a set flag (left or right) and its matched vertex
- The program proceeds in cycles of four phases:
  - Each unmatched left vertex sends a message to its neighbors and then votes to halt
  - Each unmatched right vertex grants one request and denies others and then votes to halt
  - Each unmatched left vertex accepts one grants it receives
  - An unmatched right vertex receives at most one acceptance message
A semi-cluster in a social graph is a group of people interacting frequently with each other and less frequently with others.

- **Input**: a weighted, undirected graph
- **Output**: at most $C_{max}$ semi-clusters
- Each vertex $V$ maintains a list containing at most $C_{max}$ semi-clusters, sorted by score
- In superstep 0, $V$ enters itself in that list as a semi-cluster of size 1 and score 1, and publishes itself to all of its neighbors
- In subsequent supersteps, $V$ adds itself to received semi-clusters, sorts them by score and propagates best ones to neighbors, and at last updates its list
- The algorithm terminates either when the semi-clusters stop changing or when the number of supersteps reaches a user-specified limit
Experiments

- Use the single-source shortest paths implementation
- Conduct experiments on a cluster of 300 multicore machines
- Measure running time with checkpointing disabled
- Measure how Pregel scales with increasing worker tasks
- Measure how Pregel scales with increasing number of vertices
- Use both binary trees and log-normal graphs
Experiments

![Runtime vs. Number of Worker Tasks](image-url)

- X-axis: Number of worker tasks
- Y-axis: Runtime (seconds)

The graph shows a decrease in runtime as the number of worker tasks increases.
Experiments

The graph shows the relationship between the number of vertices and the runtime in seconds. The runtime increases linearly as the number of vertices increases from 5G to 50G.
Experiments

Runtime (seconds)

Number of vertices

100M 200M 300M 400M 500M 600M 700M 800M 900M 1G
Pregel is a scalable, general-purpose system for implementing graph algorithms in a distributed environment.

Run a program in supersteps in which vertices do computation and send messages to others for the next superstep.

The API is intuitive, flexible, and easy to use.

Future work:
- Spill data to local disk
- Topology-aware partitioning and dynamic re-partitioning
BSP - straggler problem
- A small number of threads (the stragglers) take longer than the others to execute a given iteration
- Asynchronous Parallel Model?

Load Balancing
- Graphs have power-law degree distribution
- Does topology-aware graph partitioning suffice?
- What do we need to consider to implement dynamic re-partitioning?

How does Pregel deal with Master failure?