Pregel: A System for Large Scale Graph Processing

Malewicz et al.

Presentation by Yosef E Mihretie

How do I feel about presenting one of the few papers from Google not written by Jeff Dean and one among a few in this class not written by Julian Shun?

How do I feel about presenting one of the few papers from Google not written by Jeff Dean and one among a few in this class not written by Julian Shun?

Motivation

- Graphs are widely used for modeling problems in many different domains
- Sizes of these graphs today are gigantic and growing - billions of vertices and trillions of edge for the biggest once today
- Can not be handled on single commodity PCs. Powerful single node machines are expensive and might not support graphs in the near future => distributed memory parallel computing is a good solution
- Pregel is a distributed memory large-scale graph computing framework designed for directed graphs
- Pregel is scalable, fault-tolerant and general purpose

Model of computation

- Based on Valiant's Bulk Processing Model - vertex centric and message passing interface based
- Each vertex keeps a value for itself and for each of its outgoing edges. It can also exchange messages with other vertices.
- Computation divided into iterations called supersteps
- In superstate S, a vertex V can:
    - Receive messages sent to it in superstep S-1
    - Send messages that will be delivered in S+1
    - Mutate graph topology, which will be effective in S+1
    - Modify its state or that of its outgoing edges

Model of computation

- These are accomplished by invoking a user defined function on active vertices

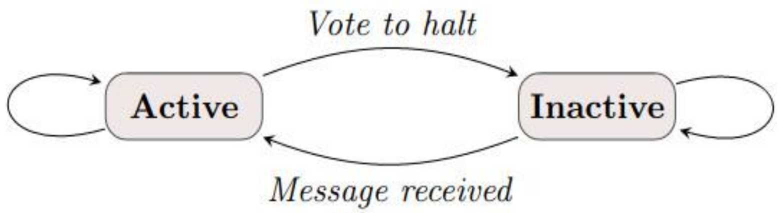

Figure 1: Vertex State Machine

- Initially, all vertices are active. Computation stops when all vertices are inactive and there are no messages in transit.

Model of computation

Example:

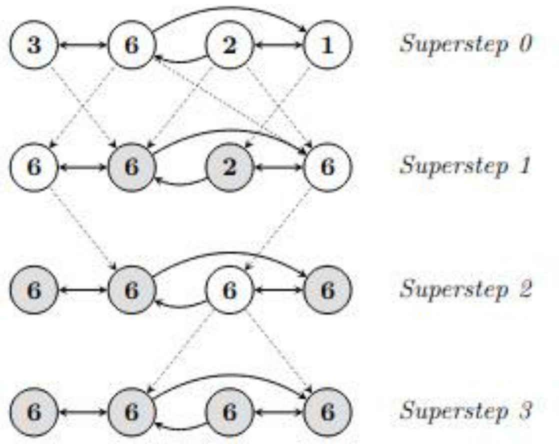

Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

API

- Vertex class

```
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();
};
```

Figure 3: The Vertex API foundations.

- User subclasses this and overrides Compute()

- Messages can be sent to any vertex if identifier is known

API - Combiner and Aggregator

Combiner class:

- For some algorithms, only some commutative and associative combination of the messages matters
- To reduce network traffic, users can subclass the Combiner class and override the Combine() method to define how messages can be combined for these algorithms

Aggregator class

- Used for global communication, monitoring and data
- Operates on values provided by vertices
- Example use - Delta-stepping shortest path

API - Topology Mutations

- Vertices can issue a request to add or delete a vertex or an edge

- Partial ordering: Edge removals > vertex removals > vertex addition > edge addition > Compute()

- Other conflicts are handled by randomly choosing an operation among conflicting once by default

- Custom handlers can also be provided by user

Implementation Details

- Executed on a cluster of 1000s of commodity PCs
- Cluster management system for scheduling jobs, allocating resources, moving tasks between PCs
- Name service, persistent storage (GFS, BigTable) available
- Vertices split into partitions and partitions allocated to Worker machines
- Partitioning method can customized
- A master computer coordinates worker activity

Fault tolerance

- Commodity PCs are vulnerable to failure
- Fault tolerance is achieved by checkpointing to a persistent storage
- Master pings workers. If no response in a certain time, worker considered dead and partitions reassigned to other workers
- Recovery is done by reverting the entire operation to the last checkpoint
- Optimization: checkpoint exchanged messages and recover only the partitions of dead workers

Worker Implementation

- Each assigned partition runs in a thread
- Vertex state and incoming message require two queue each - one for this superstep and one for second
- Loop through vertices in a partition, invoke Compute
- Put messages being sent in an outgoing buffer if receiver in another machine, or directly put in the buffer of the receiver queue if in the same machine

Applications - PageRank

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

Figure 4: PageRank implemented in Pregel.

Applications - SSSP

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

Figure 5: Single-source shortest paths.

```
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);
  }
};
```

Figure 6: Combiner that takes minimum of message values.

Experiments

- Run SSSP on a 300 multicore commodity PCs cluster on a 1B vertex, 1B binary graph

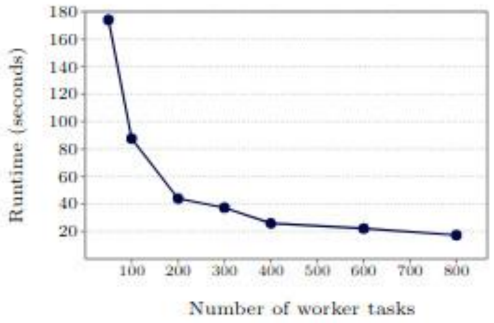

Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines

Experiments

- Run SSSP on a 300 multicore commodity PCs cluster on binary graphs

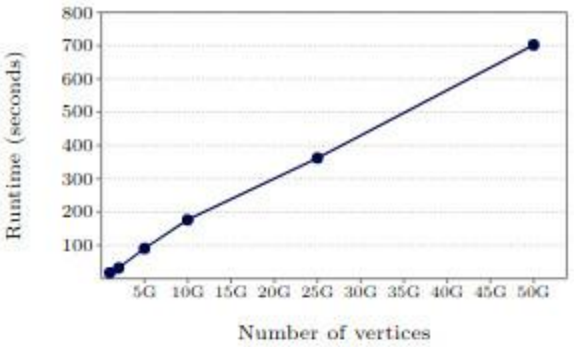

Figure 8: SSSP—binary trees: varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

Experiments

- Run SSSP on a 300 multicore commodity PCs cluster on random graph that use a log-normal distribution of outdegrees,

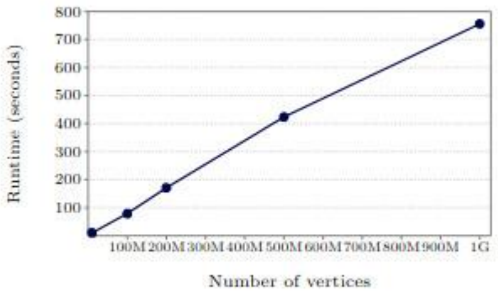

Figure 9: SSSP—log-normal random graphs, mean out-degree 127.1 (thus over 127 billion edges in the largest case): varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

142

Strengths and Weaknesses

Strengths

- API is easy to use and reason with
- Scalability
- Fault Tolerance
- Generality

Weaknesses

- Generality not rigorously established
- Experiments insufficient

Future Work

- Formulating generalizability rigorously sounds appealing
- Extensive experiments to establish comparative advantage over external computation on commodity PCs
- Cost-benefit analysis on similar algorithms in shared-memory frameworks on advanced single node machines
- Experiments to establish where the bottlenecks are in Pregel
- Partitions methods and dynamic partitioning as graph changes

Thank you!