Introduction to Pregel

Paper Authored By: Malewicz et al.

By: Shwetark Patel
What is Pregel?

- Framework for processing and modelling algorithms on large graphs
- Examples of large graphs
  - The Web
  - Transportation networks
- Created by Google Researchers
The Problem

- Running algorithms on large graphs is difficult
  - Distribution over machines leads to issues with machines failing and locality
  - Algorithms are often complex and take many lines of code to write, potentially leading to bugs
  - Current frameworks (such as MapReduce) are ill-suited for graph algorithms
Pregel Basics

- Each vertex is assigned a vertex identifier and associated with a user-defined value
- Pregel computation is defined by a sequence of “supersteps”
- All computation within a superstep is done in parallel
- At each superstep, each vertex performs a user-defined function
- Each vertex can see messages sent to it at the previous superstep and send messages to other vertices for the next superstep
Pregel Basics 2

- Vertices can change the topology of the graph during each superstep
- All vertices are **active** at the beginning
- At any superstep, a vertex can **vote to halt** and then it stops computation for all future supersteps (though it can be reactivated with a message)
- Computation is finished when all vertices vote to halt
- The output is simply the set of values outputted by each of the vertices
C++ API Basics

- Pregel users implement algorithms through the C++ API
- One must create a subclass of the predefined Vertex class
- `Compute()` -- Executed by each vertex at each superstep
  - `GetValue()` -- Get the value associated with the vertex
  - `MutableValue()` -- Modify the value associated with the vertex
Message Passing

- Vertices can pass messages to each other, that can be read at the next superstep
- The order that messages will be delivered is not guaranteed; it is only guaranteed that they will be sent
Combiners

- Sometimes, not all messages individually are important to a vertex. Only the messages in aggregate are important.
  - For example, perhaps a vertex only needs the sum of all incoming message values
- We can make our program more efficient by writing a Combiner that combines messages delivered to a vertex
  - We should only write combiners for associative and commutative operations (because we can’t assume the ordering of messages or how they’re grouped)
Aggregator

- Each vertex provides a value to an aggregator in a superstep, and then the aggregator combines all values together using an operator and provides the result for the next superstep.
- Possible use case: Select a “special” vertex by providing all vertex IDs to the aggregator and choosing the maximum one.
- Sticky aggregator: Combine values from all previous supersteps rather than just the last one.
Topology Mutations

- Multiple vertices can try to change the topology of a graph in a single superstep
- Removals are performed before additions, with edge removal before vertex removal and vertex addition before edge addition
- Other conflicts are handled with user defined handlers
Pregel Architecture

- Each vertex is assigned to one of many partitions
  - The default assignment function is $\text{hash}(\text{vertexID}) \mod N$, where $N$ is the number of partitions
- Many copies of the Pregel program are executed on a cluster, with one copy acting as the master
- The master is not assigned a partition like each of the workers are -- rather, the master just coordinates worker activity
Pregel Architecture 2

- The master determines the number of partitions and assigns machines to partitions
  - The user can also control the number of partitions
- The master tells workers when to perform supersteps
- The master can ask workers to save their partitions at the end
Fault Tolerance

- Done through checkpointing
- The master asks workers to save partitions to persistent storage at the beginning of supersteps
- Worker failures detected using ping messages
- Partitions from failed workers are reassigned (a few previous supersteps may need to be repeated depending on when the worker failed)
Worker Implementation

- Worker stores the partition in memory
- Worker stores an incoming message queue and a flag denoting whether the vertex is active (in fact, it stores two copies of these: one for this time-step and one for the next).
- When sending a message to another vertex, if the vertex is on another machine, the message is added to a buffer for delivery.
- When the buffer size becomes large enough, it is emptied and all messages are sent to the destination vertex.
Master Implementation

- The master coordinates the worker’s activities
- The master also maintains various statistics about the graph
  - Size of the graph
  - Number of active vertices
  - Message traffic per superstep
Application: PageRank

- At each superstep, each vertex sends its (Current Page Rank) / (Number of outgoing edges)
- Each vertex sums up incoming messages and uses a formula to determine its new PageRank
- This process is repeated 30 times (though in reality page rank should go until convergence)
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();
      }
   }
};
Application: Shortest Paths

- At each superstep, each vertex receives potential minimum distance updates from its neighbors (from the previous superstep) and updates its value if necessary.
- If an update is indeed made, it sends out potential updates to its neighbors.
- The algorithm terminates when no updates are made.
Experiments

- Experiments were run using a Single Source Shortest Path implementation in Pregel
- Researchers found that runtime scaled well with an increasing number of worker tasks
- Pregel also produced satisfactory (but not optimal) results given the minimal coding effort put in
Experiments

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

- **Strength**: Good results with minimal coding effort, fault tolerant, and flexible
- **Primary weakness**: Might be slower than optimized implementations
Future Work

- Potentially scaling the framework to even larger graphs
- Assigning vertices to machines to minimize the number of messages sent between different machines
Discussion Questions

● What are some potential attributes of algorithms that would prevent Pregel from being efficient in modelling them?
● Compare Pregel to other similar frameworks, such as Ligra