Pregel: Distributed System for Large Graph

Qiyuan Qiu
School of Arts and Sciences
Computer Science
University of Rochester

***Abstract*—Large graph is ubiquitous. Examples include Facebook's relationship graph, Google's PageRank algorithm. The scale of these graphs are the main challenge. They are usually billions of vertices and trillions of edges. Targeting at this problem, this paper discusses an abstraction of large scale graph programming – Pregel. It focuses on vertices in the graph. Programs are executed in iterations. During each iteration, each vertex in the graph can do the following things: receive message from the previous iteration; send message to other vertices; modify its own state and the state of its outgoing edges; change graph topology.**

I. Introduction

Graph is a common model for various practical issues. Typical examples are "similarity of newspaper articles, paths of disease outbreaks, or citation relationship among published scientific work". Popular algorithms on graphs shortest path, clustering and PageRank, minimum cut etc.

Due to the poor locality of memory access, very little work per vertex and changing degree of parallelism over course of execution, processing large graph is hard. The existing approaches to run an algorithm on a large graph is typically the following:

- Setting up a customized hardware environment specifically for the algorithm that is going to run on the given graph
- Using existing programming abstraction for large graphs like MapReduce. However, MapReduce was not designed for this particular algorithm. To make the algorithm work using MapReduce often requires a lot of tunning.
- Choosing from some single machine algorithm library for the algorithm. This approach usually has scalability issues.
- Choosing from some existing parallel graph systems. This approach does not have a mechanism for fault tolerance.

All of these are not ideal way to program algorithms on large scale graphs. However Pregel is the abstraction that is highlighting fault-tolerance and scalable to address this challenge.

Pregel computations is based on global iterations or *superstep*. Synchronizations happen within superstep. Iterations continue until termination conditions are met. For each superstep, the Pregel calls a user defined function attached to each vertex. The function can do the following things: It can read message from the previous superstep; send message to the next superstep; modify the state of itself and its outgoing edges. Messages can be sent to any vertex (if that it receives the identifier of other vertices)

This abstraction is similar to MapReduce in that clients focus on local computations(on vertices) work on their computations independently, and the server combines these results to a larger dataset. Because the computation is independent with each superstep and all communications happen between supersteps, this abstraction is easier to analyze and implementation is free from deadlocks and data races issues exist in most other common asynchronous systems.

II. Model of Computation

The input to the Pregel abstraction is a directed graph. In that graph, each vertex is given a *vertex identifiler*. It is used to refer to each vertex. Each vertex has a value that could be modified by user associated with it. Edges in this model is attached to vertices. It could also be assigned values.

While execution, Pregel takes in a graph and keep running from superstep to superstep until terminate conditions are met. At that point it outputs a resulting directed graph.

Within each superstep, the vertices compute the same user defined function. The algorithm run on the large graph is encoded in this user defined function. All these vertices run in parallel.

As is described before, each vertex can do the following things: 1. modify its own state. 2. modify its outgoing edges' states. 3. receive message sent to it from the previous superstep. 4. send messages to other vertices whose identifiers are known to this vertex. 5. mutate the topology of the graph (can remove edges and itself). Edges in the graph dose not have associated user defined function.

The termination of Pregel depends on two conditions. The first is if there exist any vertex that is active. If there is still active vertex, Pregel will not terminate. The second condition is if there are any messages unsend. If there is message has not been sent, the abstraction will not terminate. A graph from the original Pregel paper [**?**] illustrate the state machine for the active and inactive state transition for each vertex.

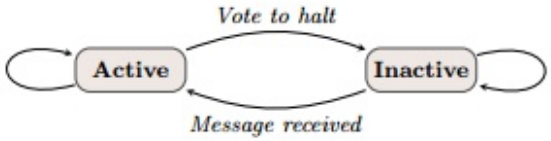

Fig. 1. Vertex State Machine

MapReduce can do what Pregel does but it is much more complicated to program. Each superstep can be treated as one MapReduce operation. However, Pregel is easier because the communication overhead is smaller than MapReduce which passes all the state from one iteration to next.

III. The C++ API

The implementation consists of creating a *Vertex* class in C++. The user needs to write a Compute() method. This method is the user specified function for each vertex and will therefore be run within each superstep. Compute() can look up vertex's value by calling GetValue() and modify the vertex's value by invoking MutableValue(). As for the value for attached edges, the vertex can change their value by calling GetOutEdgeIterator(). All state changes caused by above mentioned functions take effects immediately for all these options are locally constrained.

1) Message Passing: Each vertex can communicate with other vertices by sending messages. A vertex can send any number of messages within a superstep. The message type is specified by user in the Vertex class.

Messages sent in superstep S are guaranteed to arrive at designated vertex in superstep $S+1$. The order of message may not be assured but there will be no duplication of messages.

When there is no such destination vertex, a user defined function will be invoked to handle this exception.

2) Combiners: For some graph algorithms, overall message sent between vertices could be dramatically reduced with the help from users. This is achieved through a Combine() method provided to user. For instance, when a vertex receives multiple messages containing only integers and only the sum of these integers matter. Then Pregel can combine all these integers into their sum so that reduce the communication workload to sending one message containing only the resulting sum. However, this is not enabled by default for only commutative and associative operation can be used for Combiners.

3) Aggregators: This is the mechanism Pregel provides for global communication. Very much like global variable. Within each superstep S, each vertex can provide a value to an aggregator, Pregel will use a reduction operator to produce a sum which is going to be used for the $S + 1$ superstep. Aggregators should be both commutative and associative.

4) Topology Mutations: Algorithms like minimum spanning tree can change the topology by removing edges from the starting graph. This sort of modification to the graph is achieved through Compute() function too. A vertex can only delete itself and its outgoing edges.

A. Implementation

Pregel was designed for Google clusters. Clusters consists of commodity PCs interconnected with high bandwidth.

1) Basic architecture: A graph is partitioned into different regions. Each region is consisting of a set of vertices and their edges. This partition is solely based on vertex identifiers. This is equivalent to saying that we know in advance which region a particular vertex belongs to. Even if some vertex has not yet existed. This partition process can be replaced by user defined process. For example user can design a process in which close vertices are assigned to same machine.

Typical execution of Pregel is like the following:

- Pregel runs on a cluster of machines. Every machine runs the same copy of the program. One one them is assigned to be the master and others workers. Workers will resolve the position of the master and sends message to the master. The master is not assigned to any region of the graph. It just sits there and coordinate all the workers.
- The master decides how many regions there should be and assign vertices to machines. Allow multiple regions on one worker usually increases parallelism.
- The master assigned portion of inputs to workers. Usually the input is decomposed into portions based on file boundaries, which has noting to do with the partition of graph. Therefore workers can read in input portions that does not belong to its region. When input belongs to the workers regtion, its part of region is updated immediately, otherwise when the input portion belongs to other workers, messages to designated worker will be sent. When input finalizes, all vertices are marked active
- The master tell workers to start superstep. Terminate until all vertices are inactive and no messages in transit
- Upon termination, the master tells the workers to save the result.

2) Fault tolerance: Because commodity PCs are used as the computing cluster. It is very likely some workers will break down during execution. This break down is to be determined using a "pinging" technique. With a fixed interval of execution time, the master sends a ping message to workers. If a worker does not receive a message for some course of time, it terminates its process. If the master does not hear back form some worker for a fixed amount of time, the master mark the worker as down.

When failure happens, the master reload the graph region that fails to the start of the current superstep and redo all the computation from there.