Chapter 1

Pregel

Contributed by Justus Isaiah Hibshman

1.1 Background

Pregel [2] was developed by Google as a system to speed up their gigantic graph computations. Existing systems either handled the structure of graphs poorly (e.g. mapreduce), or were designed for single machines, or didn't have fault tolerance built in. Hence Pregel.

Pregel is built according to the bulk synchronous parallel model [1]. It's a system for distributed graph processing.

Pregel's code is proprietary; however, open-source implementations such as Apache Giraph are available (giraph.apache.org).

1.2 Expressing Graphs

Graphs in Pregel are directed. Vertices are indicated with unique string identifiers, and with every vertex is a record of all its outgoing edges.

Every vertex and every edge have associated properties defined by overloading the corresponding base classes. In the Google context, this was either a simple value or a "protobuffer" (a Google data serialization mechanism).

1.3 Syntax

Pregel is written in C++. Writing programs consists of overloading various base classes. In particular, every program requires an overloading of the vertex class. For example, here is the basic 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();
};
```

The meanings of these various values and functions are explained in "Key Graph Primitives."

1.4 Key Graph Primitives

Programs in Pregel perform all the computation "at" the vertices. Conceptually, everything happens in a series of "supersteps." At each superstep, vertices receive messages from the previous superstep, do some processing, and then (optionally) pass messages to other vertices to be received at the next superstep.

Thus, the main function to implement is the vertex class's compute() function. Compute has various primitives available to it:

1. vertex_id() gives the id of the vertex computing.
2. superstep() gives the current superstep number.
3. GetValue() and MutableValue() allow access and change to the data stored at the vertex.
4. GetOutEdgeIterator() allows iterating over all edges emanating from the vertex.
5. SendMessageTo() allows a vertex A to send a message to any vertex B it has an identifier for. Typically this is just the vertices access-able from GetOutEdgeIterator(), but depending on the user's program a vertex may know of others as well.
6. VoteToHalt() is a vertex's way of stating that it is done. Once all vertices vote to halt, the program is done. If a vertex has voted to halt, it does no more computation until a superstep when it receives a message, at which point it "wakes up" and its vote to halt is cancelled.
7. Combiners. Programmers can implement "combiners" for efficiency purposes. They take multiple messages and combine them into a single message. This can be used to save network bandwidth when messages are being passed from one machine to another. Whether or not messages will be combined is not guaranteed, and the order in which they are combined is not guaranteed. They are simply for efficiency. Combiners must be associative and commutative.
8. Aggregators. Aggregators are similar to combiners. Every vertex is able to send a message to an aggregator, and the result of the aggregation (combination) is available to all vertices at the next superstep.

1.5 Execution Model

Vertices are sharded over different machines. In every superstep, a machine executes the compute() function for every vertex its responsible for. Messages are passed to the machines on which vertices

are stored. No computation for superstep S + 1 begins until all vertices have finished superstep S. This ensures that all messages from the previous superstep are available at the start of the current.

Fault tolerance is handled by checkpointing and re-playing if a machine fails.

1.6 Examples

The following is an example taken from the original paper of an implementation of 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();
    }
};
```

To explain the above in English: The source vertex is initialized with a value of 0 and every other vertex is initialized with a value of infinity. Every message contains the length of a path from the source vertex to the message-recipient vertex. If a vertex receives a message containing a path length less than its value, it updates its value to the new path length and transmits updated path lengths over all its outgoing edges in case it's part of a shortest path to one of them. Vertices vote to halt every superstep. Thus the program continues only if a vertex sent a message.

Below is an example combiner from the original paper to make the shortest path program more efficient:

```
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);
    }
};
```

It simply takes the minimum of the messages since that's what the vertex compute() function is looking for anyways.

1.7 Conclusion

As a final note: Though Pregel may seem limited in the "kinds" of computations it could perform, it is actually quite versatile. In the original Pregel paper [2] the authors discuss the fact that every algorithm they wished to write for graphs they were able to write in Pregel. For instance, they discuss single-source shortest path, page rank, and bipartite matching.

Bibliography

[1] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[2] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.