Gopher User Guide

Overview

Gopher is a distributed and scalable programming framework which enable running fast analytics on large time series graphs.Large scale graph analytics is a hot topic both in academia and industry.People have done significant efforts for development of scalable algorithms, programming abstractions and execution frameworks. Popularity of Map reduce programming model and its applications in big-data cause lot of traction which lead to running many graph analytics on map-reduce based programming frameworks. Recently google pregral and apache giraph programing model gain more traction in graph analytics community due to ease of programming and performance benefits compared to other data parallel models like map-reduce.

Pregal/Apache Graph provides a vertex centric , Bulk Synchronous Parallel(BSP) programming abstractions to user where processing is done at vertex level in parallel with several iterations called super steps. Messages can be passed between vertices in the graph in between these supersteps.

Gopher compliments this Bulk synchronous parallel programming model but provides a subgraph centric programming model to the user. We have observed that this is reduce the memory pressure and barrier synchronization overhead compared to the vertex centric abstraction provided by Google Pregel. Gopher also introduce iterative BSP programming model where users can compose multiple super-iterations of BSP Jobs with shared state.

Time series graphs

In short, time series graphs can be defined as graphs over time. A Time series graph can be visualized as a series of snapshots of a time evolving graph. Given the work done on parallel graph processing in large graphs this representations allows users to exploit two dimensional parallelism.

Gopher API

Gopher Provides a Subgraph centric programming abstraction to the user. Implementation of the subgraph centric application starts with user extending the *edu.usc.goffish.gopher.api.GopherSubGraph* .

Subgraph centric API.

In subgraph centic programming model users are provided with a subgraph abstraction to program on. Subgraph is defined as a weekly connected component of a graph. Processing is done in parallel in each subgraph.Subgraph tasks can communicate with each other using the subgraph ids. This can be thought as a “Think like a subgraph” programming model with contrast to a “Think like a vertex” programming model.

Iterative BSP

Gopher extends the traditional BSP model by introducing an iterative BSP programming abstraction. This allows doing multiple iterations of BSP algorithm. We have observed that this kind of programming model make is easier to do some time series graph analytics, in which traditional BSP model allows parallel computation of large graphs while iterations can be used to do traversals over multiple time snapshots.

Following is the API user is provided with.

```
/**
 * <class>GopherSubGraph</class>  Provide the Subgraph centric api for Gopher.
 * User will extend this class and implement compute() to implement the user logic.
 */
GopherSubGraph {
        /**
        * Initialize the sub-graph by providing partition and sub-graph instances to be used by the subgraph.
        * @param partition current partition
        * @param subgraph  current subgraph
        */
        public final void init(IPartition partition,ISubgraph subgraph,
                List<Integer> partitions);
        /**
        * User implementation logic goes here.
        * To send message to a another partition use {@link #sendMessage(long, SubGraphMessage)}} see
        * {@link SubGraphMessage} for more details about the message format.
        * To signal that current logic is done processing use {@link #voteToHalt()}
        * @param messageList List of SubGraphMessage which is intended for this sub graph.
        */
        public abstract void compute(List<SubGraphMessage> messageList);
        /**
```

```
    * Get the Current super step.
    * @return
    */
    public  final int getSuperStep();
    /**
    * Signal that this subgraph finished processing. The system will come to a halt state once all the subgraphs come
    * to an halt state.
    */
    public final  void voteToHalt();
    /**
    * Send Message to a given partition. Optionally to a subgraph
    * @param partitionId target partition
    * @param message data message for that partition
    */
    public final  void sendMessage(long partitionId,SubGraphMessage message);
    /**
    * Send Message to a subgraph
    * @param message subgraph message
    */
    public final void sendMessage(SubGraphMessage message);

    /**
    * Check whether the current sub-graph is in halt state
    * @return
    */
    public final boolean isVoteToHalt();

    /**
    * Get Current Iteration
    */
    public int getIteration();

}
```

Communication between subgraphs are done by exchanging subgraph messages between subgraph tasks. Subgraph message mainly contains a subgraph id which message will be routed to and data element which contains the data that needs to be transferred.

Following is the subgraph Message API

```
/**
 * <class>SubGraphMessage</class> is the Message that user can use to send messages between partitions/sub-graphs
 * @param <T>  data type
 */
SubGraphMessage<T> {
    /**
     * Create SubGraphMessage providing the data as a byte[]
     * @param data data serialized to byte[] format;
     */
    public SubGraphMessage(byte[] data);
    /**
     * Add any application specific tag
     * @param tag application specific tag
     * @return  Current Message Instance
     */
    public SubGraphMessage<T> addTag(String tag);
    /**
     * Set the target vertex id this messages is referring to.
     * This will be not used for routing the messages.
     * But its up to the developers to use this in a application specific manner
     * @param id target remote vertex id
     * @return Current Message instance.
     */
    public SubGraphMessage<T> setTargetVertex(long id);

    public long getTargetSubgraph();

    /**
     * Set the target subgrap for message.
             * This will be used to route this message to the target subgraph by the framework.
     * @param targetSubgraph
     */
    public void setTargetSubgraph(long targetSubgraph);

    /**
     * Get the Target Remote vertex id
     * @return  target remote vertex id
     */
    public long getTargetVertex();

    /**
     * Get the application specific message tag.
     * @return message tag.
     */
    public String getTag();
    /**
     * Get the message data as a byte[]
     * @return data as a byte[]
     */
    public byte[] getData();
}
```

API Usage examples.

Following is a sample application which finds weakly connected components of a graph. This is subgraph centric version of connected component identification algorithm listed in apache giraph

This algorithm finds out the smallest vertex id of a connected component at mark that connection component with that vertex id.

Here the algorithm is very simple. Each subgraph finds the smallest vertex id for each subgraph and propagate that smallest value to its connected subgraphs. If incoming value to a subgraph is different from its current value it updates the current value and propagate the changes to its neighbours.

Following is a sample implementation of the algorithm in gopher

```
@Override
public void compute(List<SubGraphMessage> subGraphMessages) {
    if(getSuperStep() == 0) {
        rids = new ArrayList<>();
        long min = 0;
        for(ITemplateVertex vertex : subgraph.vertices()) {
            if(min > vertex.getId()) {
                min = vertex.getId();
            }
            if(vertex.isRemote()) {
                rids.add(vertex.getId());
            }
        }
        currentMin = min;
        log(partition.getId(), subgraph.getId(), min);
        for(Long rv :rids) {
            String msg = "" + min;
            SubGraphMessage subGraphMessage = new SubGraphMessage(msg.getBytes());
            subGraphMessage.setTargetSubgraph(subgraph.getVertex(rv).getRemoteSubgraphId());
            sendMessage(subGraphMessage);
        }
        voteToHalt();
        return;
    }
    boolean changed = false;
    for(SubGraphMessage msg : subGraphMessages) {
        long min = Long.parseLong(new String(msg.getData()));
        if(min < currentMin) {
            currentMin = min;
            changed = true;
        }
    }
    // propagate new component id to the neighbors
    if(changed) {
        log(partition.getId(),subgraph.getId(),currentMin);
        for(Long rv :rids) {
            String msg = "" + currentMin;
            SubGraphMessage subGraphMessage = new SubGraphMessage(msg.getBytes());
            subGraphMessage.setTargetSubgraph(subgraph.getVertex(rv).getRemoteSubgraphId());
            sendMessage(subGraphMessage);
        }
    }
    voteToHalt();
}
```