Giraph: Large-scale graph processing infrastructure on Hadoop

Qu Zhi
Why scalable graph processing?

- Web and social graphs are at immense scale and continuing to grow
  - In 2008, Google estimated the number of web pages at 1 trillion
  - At the 2011 F8, Facebook announced it has 800 million monthly active users
  - In September 2011, Twitter claimed to have over 100 million active monthly users
  - In March 2011, LinkedIn said it had over 120 million registered users

- Relevant and personalized information for users relies strongly on iterative graph ranking algorithms (search results, news, ads, etc.)
  - In web graphs, page rank and its variants

![Graph Diagram]
Example social graph applications

- Popularity rank (page rank)
  - Can be personalized for a user or “type” of user
  - Determining popular users, news, jobs, etc.

- Shortest paths
  - Many variants single-source, s-t shortest paths, all-to-all shortest (memory/storage prohibitive)
  - How are users, groups connected?

- Clustering, semi-clustering
  - Max clique, triangle closure, label propagation algorithms
  - Finding related people, groups, interests
Existing solutions

- Sequence of map-reduce jobs in Hadoop
  - Classic map-reduce overheads (job startup/shutdown, reloading data from HDFS, shuffling)
  - Map-reduce programming model not a good fit for graph algorithms
  - Disk IO and job scheduling quickly dominate the algorithm
Existing solutions

- Message passing interface (MPI)
  - Not fault-tolerant
  - Too generic

- Google’s Pregel
  - Requires its own computing infrastructure
  - Not available (unless you work at Google)
  - Master is a SPOF
Giraph goals

- Easy deployment on existing big data processing infrastructure
  - Maintaining a separate graph processing cluster is a lot of overhead for operations

- Dynamic resource management
  - Handle failures gracefully
  - Integrate new resources when available

- Graph-oriented API
  - Make graph processing code as simple as possible

- Open
  - Leverage the community
From Yahoo! to

- Yahoo! Research developed original codebase
- Entered Apache Incubator in July 2011
- New Apache team quickly formed
Giraph design

Easy deployment on existing big data processing infrastructure

- Leverage Hadoop installations around the world for iterative graph processing
  - Big data today is processed on Hadoop with the Map-Reduce computing model
  - Map-Reduce with Hadoop is widely deployed outside of Yahoo! as well (i.e. EC2, Cloudera, etc.)

Dynamic resource management

- Bulk synchronous parallel (BSP) computing model
- Fault-tolerant/dynamic graph processing infrastructure
  - Automatically adjust to available resources on the Hadoop grid
  - No single point of failure except Hadoop namenode and jobtracker
  - Relies on ZooKeeper as a fault-tolerant coordination service
Bulk synchronous parallel model

- Sequential computation on a single physical machine restricts the computational problem domain
- BSP is a proposal of a “bridging” model for parallel computation
  - High-level languages bridged to parallel machines
- 3 main attributes
  - **Components** that process and/or provide storage
  - **Router** to deliver point-to-point messages
  - **Synchronization** of all or a subset of components through regular intervals (supersteps)
- Computation is done when all components are done
- Only a model, does not describe an implementation
Why use BSP?

- A relatively simple computational model

- Parallelization of computation/messaging during a superstep
  - Components can only communicate by messages delivered out-of-order in the next superstep

- Fault-tolerant/dynamic resource utilization
  - Supersteps are atomic units of parallel computation
  - Any superstep can be restarted from a checkpoint (need not be user defined)
  - A new superstep provides an opportunity for rebalancing of components among available resources
Maximum vertex value example

- All vertices find the maximum value in a strongly connected graph.
- If 1st superstep or the set a new maximum value from incoming messages, send new value maximum to edges, otherwise vote to halt (gray vertices).
Writing a Giraph application

- Every active vertex will call compute() method once during a superstep
  - Analogous to map() method in Hadoop for a <key, value> tuple
- Users chooses 4 types for their implementation of Vertex (I ➔ VertexId, V ➔ VertexValue, E ➔ EdgeValue, M ➔ MsgValue)

<table>
<thead>
<tr>
<th>Map Reduce</th>
<th>Giraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Mapper&lt; KEYIN, VALUEIN, KEYOUT, VALUEOUT&gt; { void map(KEYIN key, VALUEIN value, Context context) throws IOException, InterruptedException; }</td>
<td>public class Vertex&lt; I extends WritableComparable, V extends Writable, E extends Writable, M extends Writable&gt; { void compute( Iterator&lt;M&gt; msgIterator); }</td>
</tr>
</tbody>
</table>
### Basic Giraph API

<table>
<thead>
<tr>
<th>Get/set local vertex values</th>
<th>Voting</th>
</tr>
</thead>
<tbody>
<tr>
<td>I getVertexId();</td>
<td>void voteToHalt();</td>
</tr>
<tr>
<td>V getVertexValue();</td>
<td>boolean isHalted();</td>
</tr>
<tr>
<td>void setVertexValue(V vertexValue);</td>
<td></td>
</tr>
<tr>
<td>SortedMap&lt;I, Edge&lt;I, E&gt;&gt; getOutEdgeMap();</td>
<td></td>
</tr>
<tr>
<td>long getSuperstep();</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph-wide mutation</th>
<th>Messaging</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addVertexRequest(MutableVertex&lt;I, V, E, M&gt; vertex);</td>
<td>void sendMsg(I id, M msg);</td>
</tr>
<tr>
<td>void removeVertexRequest(I vertexId);</td>
<td>void sentMsgToAllEdges(M msg);</td>
</tr>
<tr>
<td>void addEdgeRequest(I sourceVertexId, Edge&lt;I, E&gt; edge);</td>
<td></td>
</tr>
<tr>
<td>void removeEdgeRequest(I sourceVertexId, I destVertexId);</td>
<td></td>
</tr>
</tbody>
</table>

- Local vertex mutations happen immediately
- Vertices “vote” to end the computation
  - Once all vertices have voted to end the computation, the application is finished
- Graph mutations are processed just prior to the next superstep
- Sent messages are available at the next superstep during compute
public class SimplePageRankVertex extends Vertex<LongWritable, DoubleWritable, FloatWritable, DoubleWritable> {
    public void compute(Iterator<DoubleWritable> msgIterator) {
        if (getSuperstep() >= 1) {
            double sum = 0;
            while (msgIterator.hasNext) {
                sum += msgIterator.next().get();
            }
            setVertexValue(new DoubleWritable((0.15f / getNumVertices()) + 0.85f * sum));
        }
        if (getSuperstep() < 30) {
            long edges = getOutEdgeIterator().size();
            sentMsgToAllEdges(new DoubleWritable(getVertexValue().get() / edges));
        } else {
            voteToHalt();
        }
    }
}
Hadoop

- Open-source implementation of Map-Reduce and GFS (HDFS)
- Meant to solve “big data” challenges such as distributed grep, process log data, sorting, etc.
- Includes resource management (JobTracker)
- Not good for message passing (every message passing step requires a Map-Reduce job)
Giraph job from Hadoop’s perspective

- Giraph has its own InputFormat that calls the user’s **VertexInputFormat**
  - Hadoop will start up the workers requested by Giraph, not based on the InputSplit objects generated by the VertexInputFormat

- Giraph has its own OutputFormat that will call the user’s **VertexOutputFormat**
  - Giraph’s internal structure will write the data with the user’s **VertexWriter**
Giraph's Hadoop usage
## Watching a Giraph job in Hadoop

<table>
<thead>
<tr>
<th>Kind</th>
<th>% Complete</th>
<th>Num Tasks</th>
<th>Pending</th>
<th>Running</th>
<th>Complete</th>
<th>Killed</th>
<th>Failed/Killed Task Attempts</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>99.99%</td>
<td>101</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0 / 0</td>
</tr>
<tr>
<td>reduce</td>
<td>0.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 / 0</td>
</tr>
</tbody>
</table>

### Job Counters
- **SLOTS_MILLIS_MAPS**
  - Map: 0
  - Reduce: 0
  - Total: 4,200
- **Total time spent by all maps waiting after reserving slots (ms)**
  - Map: 0
  - Reduce: 0
  - Total: 122,679
- **Launched map tasks**
  - Map: 0
  - Reduce: 0
  - Total: 101

### Giraph Timers
- **Setup (milliseconds)**
  - Map: 3,184
  - Reduce: 0
  - Total: 3,184
- **Superstep 0 (milliseconds)**
  - Map: 0
  - Reduce: 0
  - Total: 8,329
- **Superstep 2 (milliseconds)**
  - Map: 0
  - Reduce: 0
  - Total: 22,393
- **Superstep 1 (milliseconds)**
  - Map: 0
  - Reduce: 0
  - Total: 31,609

### File Output Format Counters
- **Bytes Written**
  - Map: 0
  - Reduce: 0
  - Total: 0

### Giraph Stats
- **Aggregate edges**
  - Map: 50,000,000
  - Reduce: 0
  - Total: 50,000,000
- **Superstep**
  - Map: 3
  - Reduce: 0
  - Total: 3
- **Current workers**
  - Map: 100
  - Reduce: 0
  - Total: 100
- **Aggregate finished vertices**
  - Map: 0
  - Reduce: 0
  - Total: 0
- **Aggregate vertices**
  - Map: 0
  - Reduce: 0
  - Total: 50,000,000

### File Input Format Counters
- **Bytes Read**
  - Map: 0
  - Reduce: 0
  - Total: 0

### FileSystemCounters
- **FILE_BYTES_READ**
  - Map: 24,900
  - Reduce: 0
  - Total: 24,900
- **HDFS_BYTES_READ**
  - Map: 0
  - Reduce: 0
  - Total: 4,444
- **FILE_BYTES_WRITTEN**
  - Map: 0
  - Reduce: 0
  - Total: 3,060,291
- **Map input records**
  - Map: 101
  - Reduce: 0
  - Total: 101
Thread architecture

Map-only job in Hadoop

Thread assignment in Giraph

Master
Worker
ZooKeeper
Thread responsibilities

- **Master**
  - Only one active master at a time
  - Runs the VertexInputFormat getSplits() to get the appropriate number of InputSplit objects for the application and writes it to ZooKeeper
  - Coordinates application
    - Synchronizes supersteps, end of application
    - Handles changes within supersteps (i.e. vertex movement, change in number of workers, etc.)

- **Worker**
  - Reads vertices from InputSplit objects
  - Executes the compute() method for every Vertex it is assigned once per superstep
  - Buffers the incoming messages to every Vertex it is assigned for the next superstep

- **ZooKeeper**
  - Manages a server that is a part of the ZooKeeper quorum (maintains global application state)
Master uses the VertexInputFormat to divide the input data into InputSplit objects and serializes them to ZooKeeper.

Prior to every superstep, the master assigns every worker 0 or more Partitions according to balancer.
Work flow
Worker phases in a superstep

- Master selects one or more of the available workers to use for the superstep
- Users can set the checkpoint frequency
  - Checkpoints are implemented by Giraph (all types implement Writable)
- Users can determine how to distribute vertex partitions on the set of available workers
- BSP model allows for dynamic resource usage
  - Every superstep is an atomic unit of computation
  - Resources can change between supersteps and used accordingly (shrink or grow)
Fault tolerance

- No single point of failure from Giraph threads
  - With multiple master threads, if the current master dies, a new one will automatically take over.
  - If a worker thread dies, the application is rolled back to a previously checkpointed superstep. The next superstep will begin with the new amount of workers
  - If a zookeeper server dies, as long as a quorum remains, the application can proceed

- Hadoop single points of failure still exist
  - Namenode, jobtracker
  - Restarting manually from a checkpoint is always possible
Master thread fault tolerance

- One active master, with spare masters taking over in the event of an active master failure
- All active master state is stored in ZooKeeper so that a spare master can immediately step in when an active master fails
- “Active” master implemented as a queue in ZooKeeper

Before failure of active master 0

Before failure of active master 0

After failure of active master 0

Active Master State

“Active” Master 0

“Spare” Master 1

“Spare” Master 2

Active Master State

“Active” Master 0

“Active” Master 1

“Spare” Master 2
Worker thread fault tolerance

- A single worker failure causes the superstep to fail
- In order to disrupt the application, a worker must be registered and chosen for the current superstep
- Application reverts to the last committed superstep automatically
  - Master detects worker failure during any superstep with a ZooKeeper “health” znode
  - Master chooses the last committed superstep and sends a command through ZooKeeper for all workers to restart from that superstep
Optional features

- Combiners
  - Similar to Map-Reduce combiners
  - Users implement a combine() method that can reduce the amount of messages sent and received
  - Run on both the client side and server side
    - Client side saves memory and message traffic
    - Server side saves memory [no combine at all]

- Aggregators
  - Similar to MPI aggregation routines (i.e. max, min, sum, etc.)
  - Users can write their own aggregators
  - Commutative and associate operations that are performed globally
  - Examples include global communication, monitoring, and statistics
What do you have to implement

- Define \( \langle I, V, E, M \rangle \) type
  - Vertex Id, Vertex Data, Edge Data, Message Data

- Subclass `Vertex`
  - Override `compute()` to implement customize algorithms

- Subclass `VertexInputFormat` to read your graph
  - `TextVertexInputFormat`
  - From a text file with adjacency lists like `<vertex> <neighbor1> <neighbor2> ...`

- Subclass `VertexOutputFormat` to write back the result
  - `IdWithValueTextOutputFormat`
What do you have to implement (cond.)

- [Optional] Combiner
  - Combined on message arrived (!!)
  - `public void combine(int vertexIndex, Message originalMessage, Message messageToCombine);`

- [Optional] Partitioner
  - Hash for default implementation ( % no. of workers)
  - Range hash
  - Others? Memcached?
What do you have to implement (cond.)

- [Optional] Observer
  - Both Worker & Master
  - Pre/Post Application/SuperStep

- [Optional] VertexResolver
  - Resolve graph mutations

- [Optional] Checkpoint & Restart SuperStep

- [Optional] Out-of-core message/graph
  - DiskBackedMessageStore
  - DiskBackedPartitionStore
## Key Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache.giraph.combiner</td>
<td>Def. &amp; Impl. of Combiner</td>
</tr>
<tr>
<td>org.apache.giraph.comm.aggregators</td>
<td>Def. &amp; Impl. of Aggregator</td>
</tr>
<tr>
<td>org.apache.giraph.io (.formats)</td>
<td>Def. &amp; Impl. of InputFormat/OutputFormat</td>
</tr>
<tr>
<td>org.apache.giraph.partition</td>
<td>Def. &amp; Impl. &amp; Partitioner</td>
</tr>
<tr>
<td>org.apache.giraph.graph</td>
<td>Graph computing framework</td>
</tr>
<tr>
<td>org.apache.giraph.master</td>
<td>Graph computing framework – Master</td>
</tr>
<tr>
<td>org.apache.giraph.worker</td>
<td>Graph computing framework – Worker</td>
</tr>
<tr>
<td>org.apache.giraph.comm.messages</td>
<td>MessageStore</td>
</tr>
<tr>
<td>org.apache.giraph.comm.netty</td>
<td>Networking &amp; Communication related logic</td>
</tr>
<tr>
<td>org.apache.giraph.conf</td>
<td>Job conf parameters</td>
</tr>
</tbody>
</table>
## Key Class

<table>
<thead>
<tr>
<th>Class</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphMapper</td>
<td>Glue to Hadoop framework</td>
</tr>
<tr>
<td>GraphTaskManager</td>
<td>Main entry of computing framework</td>
</tr>
<tr>
<td>BspService</td>
<td>Zookeeper based impl.</td>
</tr>
<tr>
<td>-&gt; BspServiceMaster</td>
<td>Master role</td>
</tr>
<tr>
<td>-&gt; BspServiceWorker</td>
<td>Worker role</td>
</tr>
<tr>
<td>NettyWorkerClientRequestProcessor</td>
<td>Deal with outgoing messages</td>
</tr>
<tr>
<td>WorkerRequestServerHandler</td>
<td>Deal with incoming messages</td>
</tr>
<tr>
<td>TextVertexInputFormat/TextVertexOutputFormat</td>
<td>Basic input &amp; output</td>
</tr>
<tr>
<td>OneMessagePerVertexStore</td>
<td>Used when Combiner is provided</td>
</tr>
<tr>
<td>SendMessageCache</td>
<td>Basic Send Cache</td>
</tr>
</tbody>
</table>
Early Yahoo! customers

- **Web of Objects**
  - Currently used for the movie database (10’s of millions of records, run with 400 workers)
  - Popularity rank, shared connections, personalized page rank

- **Web map**
  - Next generation page-rank related algorithms will use this framework (250 billion web pages)
  - Current graph processing solution uses MPI (no fault-tolerance, customized code)
Page rank benchmarks

- Tiberium Tan
  - Almost 4000 nodes, shared among numerous groups in Yahoo!
  - Hadoop 0.20.204 (secure Hadoop)
  - 2x Quad Core 2.4GHz, 24 GB RAM, 1x 6TB HD

- org.apache.giraph.benchmark.PageRankBenchmark
  - Generates data, number of edges, number of vertices, # of supersteps
  - 1 master/ZooKeeper
  - 20 supersteps
  - No checkpoints
  - 1 random edge per vertex
Worker scalability (250M vertices)
Vertex scalability (300 workers)
Vertex/worker scalability

![Graph showing scalability where the X-axis represents the number of workers, the Y-axis represents total seconds, and the graph shows the relationship between the number of vertices and total seconds for different numbers of workers. The number of vertices is shown on the right Y-axis, while the total seconds are shown on the left Y-axis. The graph includes a line that illustrates the linear relationship between the number of workers and total seconds.]
Conclusion

- Giraph is a graph processing infrastructure that runs on existing Hadoop infrastructure
  - Already being used at Yahoo!
  - Lots of opportunity for new parallel graph algorithms!

- Open source
  - [https://issues.apache.org/jira/browse/GIRAPH](https://issues.apache.org/jira/browse/GIRAPH)

- Questions/comments?
  - aching@apache.org
Ongoing work

- Benchmark
  - Pagerank/SSSP of 110M Vertex and 6.7 Billion edges

- Observations
  - 27 machines: 180 sec / ss (superstep)
  - 24 machines: 84 sec / ss (superstep)
  - Great hit of network

- Lagging & Over-distributed
  - Bandwidth sensitive - Benchmark is not productive cluster
  - Message count is of no use
  - No speculative execution, SS Time = Slowest worker’s execution Time
    - Dynamic load balance is needed. - How about index?
    - Over distributed problem
Ongoing work (cont.)

- Index of custom partition
  - Local + Global cache
    - Central / Distributed Cache – Memcache / Redis
    - DHT
  - With-index vertex
    - <Partition tag>|<Vertex id>

- Node movement
  - Index update and out-of-sync
  - Home agent + Message forwarding

- Effective metrics of load balance