DS256: Assignment 1

Renga Bashyam K G
06-02-01-10-51-18-1-15590

March 14, 2019

The code template provided was extended to include the Giraph and Spark implementations of the following four algorithms:

- Page Rank with convergence
- Weakly Connected Components
- Strongly Connected Components
- Spanning Tree

1 Experimental Setup

1.1 Turing Cluster

- It has one head node with 6-core Intel Xeon E5-2620 v3 processor @ 2.40GHz and 48GB RAM.
- It has 24 nodes each with 8-core AMD Opteron 3380 processor @ 2.6GHz and 32GB RAM.
- Simultaneous multi-threading is enabled with 2 threads being allowed in each core.
- The compute nodes are connected via an L2 Gigabit (1000 Mbps) ethernet switch.

1.2 Software Details

- **OS** : Ubuntu Linux (Kernel version: 4.4)
- **Java Version** : 1.8.0
- **Hadoop Version** : 3.1.1
- **Giraph Version** : 1.3.0

1.3 Hadoop and Giraph Global Config

- **HDFS replication factor** : 2
- **YARN maximum worker memory** : 28672 MB
- **YARN minimum vcores per worker** : 1
- **YARN maximum vcores per worker** : 6

Other configurations for Spark and Giraph can be specified via the command line while submitting the job.

1.4 Datasets

All the data sets provided are downloaded from SNAP graph database [1].

I worked only with unweighted directed graph inputs which list edges line by line and have comment lines starting with "#". I implemented a custom *TextEdgeInputFormat* reader in Giraph to skip the comment lines and read the edges alone.

I used *cit-HepTh* dataset from SNAP, which is a citation network within High Energy Physics domain, to test my implementations locally. It has around 35K nodes and 400K edges.

2 Page Rank with Convergence

2.1 Giraph

The algorithm given in [2] for finding page ranks of an unweighted directed graph has been implemented in Giraph.

2.1.1 Algorithm Overview

Algorithm 1: PageRank in Giraph - Computation

```
CONVERGENCE_THRESHOLD = 0.001;
while not all vertices voted to halt do
    if Superstep 0 then
        Initialize vertex value to (1 / numVertices);
        Send (vertex value / numEdges) to neighbors;
    else
        Receive messages and sum them up;
        update new value as (0.15 / numVertices + 0.85 * sum);
        if new value less than CONVERGENCE_THRESHOLD then
            Vote to halt;
        else
            Send (vertex value / numEdges) to neighbors;
        end
    end
end
```

In the above algorithm the page rank values are initially normalized. The convergence threshold is chosen to be 0.001.

2.1.2 Possible improvements

A *message combiner* could have been used to sum up the incoming messages to achieve speed-up.

2.2 Spark

2.2.1 Algorithm Overview

Page rank being an iterative algorithm, I have tried to simulate the Pregel algorithm using Spark. The edges are stored in a PairRDD with the source vertices being keys and target vertices being values. During each iteration of the algorithm, the messages that are to be sent from each vertex is created as another PairRDD with vertices as keys and messages as values and a join between this and the edges PairRDD results in a new PairRDD R. The values R of the result of this join would be pairs of vertices and the messages they should receive in the next iteration. This scheme is followed for all the four Spark implementations.

In page rank implementation, the result R is reduced by key using the sum operation and the values are used in the next iteration to update the page ranks. The page ranks from previous iterations undergo a left outer join with the new page ranks and their values are updated with non-null values from new page rank column. The convergence is tested by computing the change in each vertex from the previous iteration and reducing the values to get the maximum change. This is compared with the threshold to test for termination.

2.2.2 Caveat - Vertices with 0 out-degree

In the spark version, the vertices with 0 out-degree are not included in the page rank computation at all (they are lost during a join operation). This won't affect the result of other vertices as they won't contribute to their page rank values. The implementation can be trivially extended to find the vertices with 0 out-degree and compute their page rank as sum of the page rank of the vertices incident on them.

3 Weakly Connected Components

3.1 Giraph

The algorithm for finding weakly connected components of an unweighted directed graph using Pregel given in [3] has been implemented in Giraph.

3.1.1 Algorithm Overview

Algorithm 2: WCC in Giraph - Computation

```
while not all vertices voted to halt do
    if Superstep 0 then
        Initialize vertex value to its ID;
        Send vertex value to neighbors;
    else
        Receive messages and find the maximum;
        Update vertex value if needed;
        If vertex value is updated, send vertex value to neighbors;
    end
    Vote to halt;
end
```

This algorithm was pretty straight-forward to implement.

3.1.2 Possible improvements

A *message combiner* could have been used to find the maximum value among the incoming messages to achieve speed-up.

3.1.3 Complexity

The number of super-steps is bounded by the diameter of the graph.

3.2 Spark

3.2.1 Algorithm Overview

To pass messages from one iteration to the next, the same strategy as the one used in page rank spark implementation is employed here. The messages received are reduced by key to find the maximum value and then they undergo a left outer join operation with the current IDs of the vertices to find if they should be updated or not. The termination condition is detected by counting the number of distinct IDs assigned to the vertices at the end of every iteration and comparing this number to that of the previous iteration and checking if they have changed.

3.2.2 Caveat - Vertices with 0 out-degree

As before, the ID of the vertices with 0 out-degree can be computed as the maximum among the IDs of the vertices incident on them.

4 Strongly Connected Components

The algorithm given in [3] for unweighted directed graphs has been implemented.

4.1 Giraph

4.1.1 Algorithm Overview

A custom class extending the *Writable* class is used to represent a vertex's value. It has a boolean variable to indicate of the vertex is active or not, the current component ID of the vertex and a list of vertices to which edges from this vertex are present in the transpose graph.

A *MasterCompute* class is implemented to synchronize across different phases of the algorithm. It has an aggregator to let the workers know the phase the computation is currently in and two other aggregators to check for the termination of a couple of iterative phases of the algorithm.

The phases involved in the algorithm are:

- *Transpose phase* : Here each vertex sends its ID to it neighbors.
- *Trimming phase* : The second superstep of the transpose phase and the first superstep in the forward traversal phase along with the trimming phase presented in the paper are done here in this implementation phase. First a vertex receives its neighbor IDs and stores them in the transpose neighbor list. Then trimming is done where vertices with in-degree or out-degree of 0 are removed. Then, the forward traversal phase is started where each vertex sends its ID.
- *Forward traversal phase* : In this phase, every vertex receives messages of their neighbor's IDs and updates their own with the maximum value. If a vertex is updated, its new ID is sent to its neighbors and a boolean AND aggregator's value is set to false. Thus, the master can know the termination is done when the aggregator is true after a superstep.
- *Backward traversal start phase* : In this phase, vertices with values equal to their current IDs start the BFS traversal within a weakly connected component and make themselves inactive. Inactive vertices vote to halt in subsequent supersteps.
- *Backward traversal rest phase phase* : In this phase, the BFS proceeds until no vertex is updated. This condition is again checked using an aggregator. The vertices reached in the BFS traversal make themselves inactive.

At the end of an iteration, after the last phase, the transpose phase starts again for the next iteration. This proceeds till all the vertices become inactive and vote to halt.

4.2 Spark

4.2.1 Algorithm Overview

The forward traversal phase proceeds as in the Spark algorithm given in the previous section for weakly connected components. The backward traversal phase proceeds as in the Spark algorithm given in the next section for spanning tree, but with multiple sources for the BFS-like traversal, each one selected from a weakly connected component.

4.2.2 Caveat - Vertices with 0 out-degree or in-degree

Vertices with 0 out-degree are automatically trimmed and are not included in the result as before. Vertices with 0 in-degree are trimmed explicity and are included in the result.

5 Spanning Tree

5.1 Giraph

The algorithm for finding the spanning tree of an unweighted undirected graph given in [4] has been implemented in Giraph.

5.1.1 Algorithm Overview

Algorithm 3: Spanning tree in Giraph - Computation

```
while not all vertices voted to halt do
    if Superstep 0 then
        Initialize vertex value (parent ID) to -1;
        If source, send vertex ID to neighbors;
    else
        if vertex value is -1, receive messages and choose one as parent;
        If vertex value is changed from -1, send vertex ID to neighbors;
    end
    Vote to halt;
end
```

This algorithm is similar to BFS. In each super-step, the vertices in the next level of the spanning tree are discovered. At the end of the algorithm, each vertex will have stored its parent's vertex ID in the spanning tree.

5.1.2 Caveat - Spanning Forest

For inputs which are disconnected, this algorithm will only find the spanning tree of the component containing the source. To extend this algorithm to find the spanning forests, the algorithm for weakly connected components should be run first and then from each components, a source should should be selected and this algorithm should be run.

5.1.3 Complexity

The number of super-steps is bounded by the diameter of the graph.

5.2 Spark

5.2.1 Algorithm Overview

To pass messages from one iteration to the next, the strategy used in page rank spark implementation is modified. After a join between edges pairRDD and the messages (in this case parentIDs) pairRDD, it is filtered to retain only those vertices which are already assigned a parent. Then the messages are extracted and reduced as before so that only vertices with parents send messages.

Now, the parent IDs of vertices are updated after a left outer join in such a way that only the vertices without a parent but with a valid message are updated. The program terminates when the number of vertices without a parent don't decrease.

5.2.2 Spanning Forest

As in the Giraph implementation, this only finds the spanning tree of the component where the source resides. This can also be extended to find spanning forests by running the weakly connected components algorithm first.

5.2.3 Caveat - Vertices with 0 out-degree

As before, the parent of the vertices with 0 out-degree can be computed as one of the IDs of the vertices incident on them.

References

[1] Stanford Large Network Dataset Collection. `https://snap.stanford.edu/data/index.html`.

[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.

[3] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[4] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.