Nick Carey
ncarey90@gmail.com ncarey4@jhu.edu

Hadoop! Triangle Counting

Describe your map/reduce algorithm for solving the three's company problem.

Describe the operation of the mapper and reducer. How does this combination solve the three's company problem?

My Mapper will read in a line of input. It will then emit every combination of the 'host' (owner of the list) and two friends in the list as a key and then 1 as a value. For example, if given:

"A B C"

as a line of input where A is the 'host' and B and C are the friends, then my Map function will emit:

"<A,B,C> -> 1", "<A,C,B> -> 1", "<B,A,C> -> 1", "<B,C,A> -> 1", "<C,A,B> -> 1", "<C,B,A> -> 1"

Note that the mapper will emit the same three elements in all different orderings. This is actually much easier explained by my Map code: (yes, I know this isn't the most efficient implementation but split.length is typically <10)

```
String line = value.toString();
String [] split = line.split("\\s+");
String host = split[0];
for(int i = 0; i < split.length; i++){
        for(int j = 0; j < split.length; j++){
                for(int h = 0; h < split.length; h++){
                        if(((i != j && j != h) && i != h) && (split[i] == host
|| split[j] == host || split[h] == host)) {
                                word.set("<"+split[i]+","+split[j]+","+split[h]
+ ">");
                                output.collect(word,one);
                        }
                }
        }
}
```

My Reducer is actually very simple. As we can see, the mapper emits every possible friend triangle present in this graph. The Reducer's job is to validate which potential friend triangles are real and which are incomplete. If a Reducer receives two or more messages with the same key, then there exists a friend triangle described by that key.   If a reducer ever only receives one copy of a given key, then that key doesn't represent a valid friend triangle.

This implementation solves the problem because to establish a friend triangle, you need to examine the friend lists of at least two members of a potential friend triangle. If A is friends with B and C, and B is friends with A and C, then A,B,C is a friend triangle. My mapper will emit all possible friend triangles given a single friend list, and my reducer will emit friend triangles emitted by at least two mappers.

What is the potential parallelism? How many mappers does you implementation allow for? Reducers?

My implementation allows for as many mappers as there are lines in the input. There are no dependencies between lines in the input. Since a key represents a possible friend triangle, there can be as many Reducers as possible friend triangles in the graph. Parallelism is obtained by splitting the input among the mappers.

What types are used as the input/output to the mapper? Motivate the transformation.

The mapper takes as input a string representing a friend list. The mapper will then output a set of keys representing every possible friend triangle containing the owner of the friend list. A single key will be a string representing a single possible friend triangle. To validate the friend triangle, another mapper processing another friend list must emit the same key. For each key, the mapper emits the integer '1' as a value. This helps the reducer keep count of the number of mappers emitting the same key by aggregating all values with the same key together. If the aggregate value is greater than one, then the friend triangle is validated.

On combiners

Why did you leave the combiner class undefined in Step 4?

A combiner is not appropriate for this problem. A combiner is a reducer that runs just outside of a mapper, and will accept key/value pairs from a single mapper. This does not work for this problem because my reducer needs key/value pairs from at least two different mappers before the reducer will emit anything at all. Therefore, a combiner in this problem would never emit anything, since it would only receive key/values from a single mapper. Potential friend triangle keys will never reach an actual reducer since its impossible for the combiner to emit anything if it is fed from a single mapper.

Generalize the concept: What sort of computations cannot be conducted in the combiner?

Any computation where the reducer depends on input from multiple mappers in order for the reducer to emit any value cannot be run with a combiner. Combiners cannot act as filters that depend on input from multiple mappers.

Analyze the parallel and serial complexity of the problem and your M/R implementation (in Big-O notation). You should assume that there are n friends list each of length l, i.e. n users that each have l friends.

What is the fundamental serial complexity of the problem? Think of the best serial implementation.

The best serial implementation would be similar to my MapReduce implementation. The serial program would scan through each friend list. While doing so, it will maintain a HashMap mapping strings representing friend triangles to integer counts. The keys in this hashmap will be the same as the keys emitted by my mapper; the values will be the same as the aggregate values maintained by my reducers. As the serial program iterates through friend lists, it will find every triple combination of the owner of the friend list and two friends. It will store this combination as a key in the HashMap and increment the value associated with this key. After scanning the entire list of friend lists, there would be a scan through the HashMap collecting each key that has a value greater than one.

Scanning through a single friend list will take approximately l*l (or l^2) steps. To examine the entire friend list, n * (l^2) steps are performed. A scan of the resulting HashMap will result in just as many operations. Therefore, the complexity is on the order of O(n * (l ^2))

How much work in total (over all mappers and reducers) does the Map/Reduce algorithm perform?

The MapReduce algorithm performs the same amount of work over all mappers and reducers.

How much work is performed by each mapper? By each reducer?

At maximum parallelism, each Mapper examines a friend list. Therefore it performs l^2 amount of work, where l is the length of the friend list.

Each Reducer maintains an aggregate count for a given key, or friend triangle. All it does is increment the aggregate value when it gets a message. Therefore the Reducer itself does constant work. The real work is done by the message passing framework in Hadoop! when it decides where to send each key.

Based on your answers to the above, describe the tradeoff between complexity and parallelism (qualitatively, you have already quantified it in the previous steps).

Parallelism has a higher overhead cost of setting up multiple processes and sending messages between processes. These overhead costs add in complexity. A serial program has much less overhead to worry about. Everything needed is contained in a single process. However, in a practical view, parallelism allows you to divide the complexity and work among parallel workers, allowing for a faster real-world execution. Depending on your available resources, its possible that trading a lower-complexity serial program for a higher-complexity parallel program is a good choice.