Assignment 3 Solutions

Detian Deng

April 1, 2015

1 Algorithm Description

In the following answers, "triad" is used to indicate a three-node undirected graph structure with at least two edges among them.

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

```
Mapper():
1. Read in one input file, and parse it as an array of integers, A
2. Set the first integer, A[1] as the key-person.
3. for i in 2 to A.length
4.   for j in (i+1) to A.length
5.     create an array of integers: B = [ key-person, A[i],A[j] ]
6.     key = Sort(B), value = 1
7.     output the <key, value> pair

Reducer():
1. Read in all <key, value> pairs with the same key,
   as a result of sorting and shuffling.
2. Sum up the values of these key pairs.
3. If the sum is greater than 1, output the three desired permutations of the key.
```

In line 3-4, the *Mapper* procedure traverses all two-friend combinations chosen from the friends list of the key-person, and line 5 stores the triad suggested by the combination in B. In line 6, the sorting of B makes it an unique identifier for that triad. Thus a single *Mapper* instance reads in one input file and outputs all the triads (triangle candidates) containing the key-person. After all mappers finish, the all output keys together will contain all the triads in the graph, and the true triangles must be a subset of these triads.

For a given key, the *Reducer* procedure sums up the times it shows up in the *Mapper*s' output. Since the data are symmetric, a true triangle must have a sum being exact 3 because its corresponding triad shows up once for each node while it is the key-person, but a triad without the third edge will show up only once when the key-person is the root node. Thus the *Reducer* can tell which triad is in fact a triangle, and output the three permutations as required.

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

The potential parallelism lies in both the mapping and reducing procedure. The maximum number of mappers allowed by the above implementation is the number of input files, i.e. the number of people(nodes), n, in the graph. The maximum number of reducers allowed is the number of all triads in the graph, which is upper bounded by $\binom{n}{3} = O(n^3)$.

c. What are the key/value schema for the input to the mapper and the output of the mapper (same as input to reducer)? Motivate the transformation.

The input data for the *Mapper* is organized as a line of space separated integers, where the first integer represents the current person's id, and the rest represent its friends' ids. The input data is read in as a $< key, value >$ pair, where the key is a LongWritable variable assigned by Hadoop and the value is the a Text variable containing the line of input.

The output of each *Mapper* is a set of $< key, value >$ pairs, where each key is a Text variable containing a line of 3 sorted and space separated integers (person id), and all the values are set at a IntWritable variable 1.

The motivation is that this transformation converts the triangle search problem into a simple "word count" problem, which is easy to solve and can be simultaneously computed for each key . In fact, the line 1-2 in this *Reducer* is the same as in the $WordCountReducer$.

2 On Combiners

a. Why was it necessary to leave the combiner class undefined in Step 4?

The *Combiner* function is used as an optimization for the MapReduce job. The *Combiner* function runs on the output of the map phase and is used as a filtering or an aggregating step to lessen the number of intermediate $< key, value >$ pairs that are being passed to the reducer. But the constraint is that each *Combiner* can only be used locally on a subset of $< key, value >$ pairs output by *Mapper* instances on the same node.

In this problem, all *key*s output by a single *Mapper* instance are distinct, so there is nothing to combine. Even if there were multiple *Mapper* instances on the same node, most *key*s would appear only once and all of them would appear at most 3 times, if the graph were not too dense, thus the intermediate $< key, value >$ pairs reduction would not amortize the cost of running *Combiner* function. Therefore we should leave the combiner class undefined.

3 Analysis

a. How many messages do your mappers output? Assuming the the Hadoop runtime has to sort these messages how much total work does the algorithm do (in Big-O notation)? You should assume that there are n friends list each of length $O(n)$.

Suppose the ith person has m_i friends, where $0 \leq m_i < n = O(n)$. Each $Mapper$ instance generates $\binom{m_i}{2} = O(n^2)$ outputs, thus with n $Mapper$ instances, the mappers generates $O(n^3)$ messages.

To sort all these output keys from mappers, a comparison-based sorting procedure (e.g. merge sort) has total work $O(n^3 \log(n^3)) = O(n^3 \log(n))$, although in this problem, each output key can be converted into a integer by filling missing digits with 0 (e.g. $9 \rightarrow 009$) and concatenate the 3 integers in the key (e.g. 1 23 456 $\rightarrow$ 001023456), to which a count sort procedure can be applied with total work $O(n^3)$. Based on the Yahoo tutorial, the standard implementation in sorting phase is a paralleled merge sort, so the total work for the Hadoop runtime sort these messages should be $O(n^3 \log(n))$.

b. How does this compare with the performance of serial implementations. Describe the tradeoff between complexity and parallelism

The naive serial implementation mentioned in the given link takes $O(n^3)$ and a faster serial implementation by computing $tr(A^3)/6$, where A is the adjacency matrix used to store the graph, takes only $O(n^{2.37})$.

As described in part (a), the mapping phase has total work $O(n^3)$, the sorting and shuffling phase has total work $O(n^3 \log(n))$, and the reducing phase also has total work $O(n^3)$, thus the overall time complexity is still $O(n^3 \log(n))$, which is larger than the naive serial implementation. But as the number of slave cores increases, each phase can achieve a sub-linear speed up.

As we can see, in order to take advantage of parallelism, sometimes we need to sacrafice the overall complexity comparing to the serial implementations. As a result, given a fixed amount (non-trial size) of data, when the number of computing units available is low, serial implementation can be faster than the parallel implementation, and as the number of cores increases, benefits of parallelism will gradually dominate its higher complexity, and the threshold number is likely to be $O(\log(n))$.