Homework #3
Parallel Programming
Three's Company

Yifan Ge

April 9, 2013

Problem 1

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

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

   - **Mapper:** For one line of input enumerating A's friends like this:

     $$A, B_1, B_2, B_3$$

     This input means A has three friends: B_1, B_2 and B_3. We output the following key-value pairs (the 1s are actually meaningless):

     $$(A, B_1, B_2) \to 1$$
     $$(A, B_1, B_3) \to 1$$
     $$(A, B_2, B_1) \to 1$$
     $$(A, B_2, B_3) \to 1$$
     $$(A, B_3, B_1) \to 1$$
     $$(A, B_3, B_2) \to 1$$
     $$(B_1, A, B_2) \to 1$$
     $$(B_1, A, B_3) \to 1$$
     $$(B_2, A, B_1) \to 1$$
     $$(B_2, A, B_3) \to 1$$

$$(B_3, A, B_1) \rightarrow 1$$
$$(B_3, A, B_2) \rightarrow 1$$

- **Reducer:** For each key we count the number of its occurrences. If one reducer gets two records with the same key, we output the key as a trio.

Every triple the mappers spit out is a potential trio. For example, the first line ((A, B_1, B_2)) in the above example means A is friend with B_1 and B_2, to make it a trio, we also need to know B_1 is friend with B_2. This can be confirmed if we see another (A, B_1, B_2), which must be spit out by a friend list starting with B_1. This is because we swap the first and the second elements in each triple and output the new triple. If B_1 is friend with B_2, and B_1 is friend with A because of the symmetry guaranteed by the input, we can get another (A, B_1, B_2) from the friend list input starting with B_1.

2. What is the potential parallelism? How many mappers does you implementation allow for? Reducers?
   Answer: As each line of the input is decomposed into triples by the mappers, and the triples are distributed to the reducers to recognize the duplicate keys, there is no data dependency. The mappers and reducers both can run simultaneously. If the input files contains n lines, at most n mappers are allowed. If the mappers output m different keys, at most m reducers are allowed.

3. What types are used as the input/output to the mapper? Motivate the transformation.
   Answer: The input of mapper is `<Object, Text>` and the output is `<Text, IntWritable>`. As the key of input is the line number, which we don't care about, so we leave its type to be Object. The value is the actual friend list, we read it in a `String`, which is `Text` in Hadoop. As each line of the output consists of a friend triple (which we output as a string) and a 1 (dummy output), so the type of key is `Text` and the type of the value is `IntWritable`.

Problem 2

On combiners

1. Why did you leave the combiner class undefined in Step 4?
   Answer: The combiner is not suitable for the algorithm is because that the two duplicate keys may appear in the outputs of different mappers, we cannot distinguish within a mapper what triples appear twice and what triples are singletons because we are using the `IntWritable` values only as placeholders. But in fact we can use combiners if the `IntWritable` values are used to count the number so we can merge the duplicate keys. However, as one same key can appear at most twice, the efficiency improvement provided by the combiner may be not fair enough comparing to the cost introduced.

2. Generalize the concept: What sort of computations cannot be conducted in the combiner?
   Answer: In general, a computation can be conducted in the combiner if it is both commutative and associative. For example, the addition computation in word count can be conducted in the combiner. In contrast, computations that the order or association matter cannot be conducted in the combiner.

Problem 3

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 friends.

1. What is the fundamental serial complexity of the problem? Think of the best serial implementation.
   Answer: For the friend list starting with i, we choose two distinct value j and k at one time. Since j and k are already friends of i, we need to check if k is a friend of j by looking at the friend list starting with j. As we do this for each of the n friend lists, and the length of each of the friend lists is l, the time complexity of the algorithm is $O(mnl^2)$, in which m is the time used to check if k is a friend of j. If we build a hash table for each friend list, and we suppose the time complexity of the lookup operation is $O(1)$, the overall time complexity of the serial algorithm is $O(nl^2)$.

2. How much work in total (over all mappers and reducers) does the Map/Reduce algorithm perform?
   Answer: Every friend list of length l is decomposed to $O(l^2)$ triples, so the total number of triples spit out by the mappers is $O(nl^2)$. To sort all the records, the time used is $O(nl^2 \log(nl^2))$. The reducers loop through all the records and output the duplicate ones, and this can be run in $O(nl^2)$ time.

The overall time complexity of is $O(nl^2 \log(nl^2) + \alpha)$ where α is time used to assign tasks to mappers and reducers and other overhead introduced by Map/Reduce.

3. How much work is performed by each mapper? By each reducer?
   Answer: Assume the number of mappers is m_1 and the number of reducers is m_2. So the work performed by each mapper is $O(\frac{nl^2}{m_1})$ and the work performed by each reducer is $O(\frac{nl^2}{m_2})$.

4. Based on your answers to be above, describe the tradeoff between complexity and parallelism (qualitatively, you have already quantified it in the previous steps).
   Answer: The time complexity of serial implementation is lower than the total running time of the Map/Reduce algorithm, but as the Map/Reduce algorithm is run in parallel, the actual running time is lower than the serial implementation. Moreover, add more mappers/reducers will also introduce more overhead.