Jason Zhang

Parallel Programming HW4

1.

a.

For my implementation of the problem, in order to find a “triangle” of friends I first parse all of the input files and take advantage of the symmetry(A friends with B implies B friends with A) property by generating all possible permutations of size 3 with increasing integer sorting on the last two elements. Each permutation’s first value will be every other value in the file excluding the first person while the second two elements are the correct sorting of the first value of the file and some other value of the file not equal to the permutation’s first value. In essence we are generating the other two possible edges of the triangle of friends from our given input. If we run this algorithm on all input files, we will generate the same types of permutations for each input and then we will count the number of duplicate keys. If the number of duplicate keys is equal to 2 then we know that two different people have generated a possible permutation for one of their triangle edges including the third person. By the symmetry property defined, that third person will be friends of the other two people and thus we have a triangle. Applying this overview algorithm to our map/reduce paradigm, we can generate as many mappers as we do input files and each mapper will generate the possible other sides of the triangle(described above) and make each permutation a key and assign 1 for the value. Now we shuffle the key value pairs to our reducers, which there could be as many reducers as we do distinct keys. In each reducer, for a distinct key, we take in all the key value pairs outputted by the mappers which have the same key and then keep a counter as we count the number of duplicate keys. If our count reaches 2, we output that permutation. After the reducing phase finishes we will have outputted all the triangles of our input data, which is what our problem required.

1

b.

Since our input data is independent of one another, we can have one mapper for each input file. Thus, if we have n input files, we could potentially have n mappers. In the same vein, our intermediate keys that our mapper generates are also independent, thus we can have each mapper process a distinct key. And thus, if we have d distinct keys we can potentially have d reducers. Therefore, as our input size increases, our map reduce can scale along with it. With this you can see our potential parallelism comes with our input in that our inputs for both the mapper and reducers are independent which means we can have optimal parallelism since our mappers and reducers can operate without running into data dependency issues.

2.

a.

A combiner acts as a mini-reducer in that the aim is to reduce the network bandwidth by trying to reduce the amount sent to the actual reducers. It does this by trying to group the outputs of each mapper before sending the key, value pair to the reducers. Thus this works especially well when there are duplicate keys outputted from each mapper. In our case however the combiner does not produce any helpful groupings as specifically the output from each mapper has distinct keys. This means that each mapper will not contain duplicates locally. Thus the combiner will not be able to aggregate any of the keys to lessen the bandwidth. If this is the case then using combiners will only add overhead to our map reduce algorithm and it would be much more efficient to just send the output directly to the reducers.

3

a.

From our mapper, since each friend list is length O(n), we output on the order of O(n^2)(from our algorithm trying to generate those permutations)messages per input. Given n input files we get n*O(n^2) which is O(n^3) messages.

Total work:

We know from above our mapper generates O(n^3) messages. During the sort, Hadoop will sort in the order of O(nlogn). Given our input size of O(n^3) we get total work after sort equal to O(n^3 log n^3) = O(n^3 log(n)). Our reducer will go through all the messages and output the triangles which will be an O(n^3) iteration (because of the number of messages the mappers generate). Thus our total amount of work will be O(n^3) + O(n^3 log(n)) + O(n^3) = O(n^3 log n).

3

b.

From a shallow point of view it is clear that the map reduce algorithm is slower than the serial implementation in that the map reduce runs in O(n^3 log(n)) where as the serial runs in O(n^3). But the map reduce algorithm has greater parallel potential and better memory usage. In terms of parallel potential, Our map reduce, since the input data is independent, can create as many mappers as input files, and as many reducers as there are keys. Each of the mappers and reducers run independently so there won't be any data conflicts and we can get a very good scalable computation with our map reduce algorithm. With the naïve implementation, the algorithm will iterate through the entire graph(stored as an adjacency matrix) and check every triple. In terms of parallelizability, as we scan through the matrix, we could try to parallelize the for loops but this is limited by the issue of possible interference due to data dependencies. Therefore, as the problem size increases, the map reduce algorithm will parallelize computation much more efficiently than the serial implementation. In terms of memory, our map reduce stores far less than the serial implementation because the serial implementation requires you to store an adjacency matrix which could be very spatially inefficient if you consider possible sparse input data(many people, not many of them are friends). Then our matrix would store many values with a value of 0(when there is no edge between two people). For our map reduce, each mapper handles a file input and outputs messages on the order of O(n^2). With my implementation, I don't store all of the triangle relations, only the other possible relations for a given apex in an input file. This is significantly less than storing a global adjacency matrix for all connections between people. By the time we get to the reducer, we are given O(n^3) messages to process and group and we ultimately aggregate and store only the distinct keys. This can potentially be far less than having the adjacency matrix too. So in general, the map reduce has the potential to allocate less memory for storage in comparison to the serial algorithm which requires a strict adjacency matrix which scales with our input size.