Parallel Programming
HW3

CHANG LIU
chang.liu@jhu.edu

Describe your map/reduce algorithm for the tweet stream processing problem.

1. **Describe the operation of the mappers and reducers in both phases. How does this combination solve the tweet stream processing problem?**

The Map operation is a parallelized process which separate the input file set into several chunks. The Map process does not have to know about the internal logical structure of the input. For each chunk there is a map task and for each individual map task, it will open a new output writer per configured reduce task. Then the mapper parses the input file and generates key-value pairs. As key-value pairs are read then they are passed to the configured Mapper. The user supplied Mapper does whatever it wants with the input pair and output with key-value pairs of its own choosing. The output it generates must use one key class and one value class.

The Reduce operation takes in the output of mapper and its input is scattered in many files across all the nodes where map tasks ran. First all the data will be gathered to the local file system. The file is then merge sorted so that the key-value pairs for a given key are contiguous. This makes the actual reduce operation simple: the file is read sequentially and the values are passed to the reduce method with an iterator reading the input file until the next key value is encountered. At the end, the output will consist of one output file per executed reduce task.

In the tweet stream problem:

In the phase 1, the input to the Map/Reduce program is a twitter stream that each line of the input files is a tweet of a user. Line starts with the username and followed by the detail of the tweet with hash tags start with #.

The phase 1 mapper will handle the input from the files that it reads each line of the files with a tokenizer to find out all the tokens in the line. The first token of the line will be the username, and all the following tokens will be the details of tweets. The mapper will filter out all the hash tags from each tweet as values and output them in the format of <user1, tag1> , <user1, tag2> …
For example, if the input line is:
"user1 A B C #D E F #G H #D"
The output will be:
<user1, #D>, <user1, #G>, <user #D>
This output will then become the input of phase 1 reducer.

In the phase 1 reducer operation, we use phase 1 mapper's output as input and each reducer will operate on the input stream with the same key (username). It then reads all the values (tags) of that key and connects the values together while eliminate the duplicates. The duplicates are eliminated by scan the whole value list (tag list) when adding any new values (tags), we only add tags that are not exist in the list.
For example, if the input of the reducer is:
<user2, tag1>, <user2, tag3>, <user2, tag1>, <user2, tag4>
Then the output of the reducer will be:
<user2, tag1 tag3 tag4>

In the phase 2 mapper, it takes the output from phase 1 reducer as input. The input format is similar to the input of phase 1 mapper, whereas we only have username and hash tag in each line. Then, the mapper will find all the possible combinations of the pairs of the hash tags in the input file, output the pair as key and username as value. When creating the key (hash tag pair), the mapper will sort the tags in each pair in alphabetical manner, so tag pair <ABC BAC> and <BAC ABC> will all become <ABC BAC>, thus we could eliminate the duplicates.
For example, if the input of the mapper is:
<user1, tag1 tag2 tag3>
Then the output will be:
<tag1 tag2, user1>
<tag1 tag3, user1>
<tag2 tag3, user1>

In the phase 2 reducer, the input is the output from the phase 2 mapper. The operation on the reducer will simply be connecting all the values (username) for the same key (tag pairs) and output those key pairs which has appeared more than 1 once, this means more than 1 user have comment on the same key. So we have the list of the users who had tweeted about the same hash tag pairs.
For example, if the input of the reducer is:
<tag1 tag2, user1>
<tag1 tag2, user2>
<tag1 tag2, user3>
<tag1 tag3, user2>
<tag1 tag3, user1>
<tag2 tag3, user2>
Then the output will be:
<tag1 tag2, user1 user2 user3>
<tag1 tag3, user1 user2>

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

The potential parallelism exists in both mapper and reducer stage, since mapper and reducer could be operated on multiple nodes and the results are collected from nodes after the calculation.

For PHASE ONE:
Let's say the number of input files is n, then n mappers will be running in parallel in maximum. And number of reducer will be the total number of the #tags, which is n*t.

For PHASE TWO:
Let's say the number of input file is n2, then n2 mappers will be running in parallel in maximum. And number of reducer will be the total number of the #tag pairs, which is

$$\frac{nt(t-1)}{2}$$

3. **What types are used as the input/output to the mappers? Motivate the transformation.**

PHASE 1 Mapper
Mapper<Object, Text, Text, Text>
The input is <Object, Text> as we will use the files as input
The output is <Text, Text> as we need to output <username, #tag> which are all text format:

Input Schema: K = {}, V = {username, tweet}
Output Schema:
K = username, V= #tag1
K = username, V= #tag2
K = username, V= #tag3
….
⇨ K = {username}, V= {#tag}
The phase one mapper read the input file and output a hash table taking the first token in each line as username, and the rest tokens in the line as value where token start with #.

The phase one reducer then connects all the tags into one value for each user.

PHASE 2 Mapper
Mapper<Object, Text, Text, Text>
The input is <Object, Text> as we will use the result files from reducer one as input
The output is <Text, Text> as we need to output <#tag1 #tag, username> which are all text format:

Input Schema: K = {username}, V = {list of #tags for each username}
Output Schema:
K = {#tag1, #tag2}, V= {username}
K = {#tag1, #tag2}, V= {username2}
K = {#tag3, #tag4}, V= {username2}
…
⇨ K = {#tagX, #tagY}, V= {username}

The phase two mapper get a input list of username followed by the hash #tags the user tweeted. Then the mapper will list all the possible pairs of the hash #tags tweeted by that user as key, and the username as value.

The phase two reducer then connects all the usernames into one value for the same #tag pairs (key).

On combiners

1. Why did you leave the combiner class undefined?

When the map operation outputs its pairs they are already available in memory. For efficiency reasons, sometimes it makes sense to take advantage of this fact by supplying a combiner class to perform a reduce-type function. If a combiner is used then the map key-value pairs are not immediately written to the output. Instead they will be collected in lists, one list per each key value. When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

For the mapper’s output, the key is the username for phase one and #tag pairs for phase two, and for same key the value is different. No commutative or associative function can be performed here. So the combiner is useless here, so it is undefined.

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

The primary goal of combiners is to optimize/minimize the number of key value pairs that will be shuffled across the network between mappers and reducers and thus to save as most bandwidth as possible. Combiners can be used when the function you want to apply is both commutative and associative. Those sorts of computations that requires further comparison or requires shares data as an intermediate process cannot be conducted in the combiner.

Analyze the parallel and serial complexity of the problem and your M/R implementation (in Big-O notation). You need to parameterize this by the number of users (n) and the number of tweet topics (t). You can assume that there are a constant number of tweets per user. This is a simplifying assumption, but the analysis can be done without it.

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

For the phase one Mapper: it will take n times to get the username for n users and t times to pick all the #tags from input tweets for each user, thus the total time is O(nt)

For the phase one Reducer: in the for loop, we need to iterate t times to scan all the #tags, and for each scan, we need another while loop to eliminate the duplicate, thus the time complexity for the for loop is $O(t^2)$. We need to do this reduce for n users, so the overall time Complexity is $O(nt^2)$.

For the phase two mapper: the first for loop takes t times to complete, the second for loop takes $O(t^2)$ to complete. The mapper will need to iterate for n users, so the overall time complexity is $O(nt^2)$.

For the phase two reducer: the for loops will take n times to complete for $\frac{t(t-1)}{2}$ tweet pair, thus it has an overall time complexity of $O(nt^2)$

So the total serial complexity of the problem is $O(nt^2)$.

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

Work for the phase 1 mapper is $n(t+1)$
Work for the phase 1 reducer is nt^2

Work for the phase 2 mapper is $n\left(\frac{t(t-1)}{2}+1\right)$

Work for the phase 2 reducer is $n\left(\frac{t(t-1)}{2}+1\right)$

So the work in total is:

$n(t+1)+n\left(\frac{t(t-1)}{2}+1\right)$ for mappers, complexity $O(nt^2)$.

$nt^2+n\left(\frac{t(t-1)}{2}+1\right)$ for reducers, complexity $O(nt^2)$.

Thus the complexity is $O(nt^2)$.

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

Assume we have M mappers and R reducers:

Work for each mapper:

$$\frac{\mathrm{n(t+1)+n}\left(\frac{\mathrm{t(t-1)}}{2}+1\right)}{M}, \text{ complexity } \frac{O(nt^2)}{M}$$

Work for each reducer:

$$\frac{\mathrm{nt^2+n}\left(\frac{\mathrm{t(t-1)}}{2}+1\right)}{R}, \text{ complexity } \frac{O(nt^2)}{R}$$

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

If parallelism increases, then the time complexity decrease in some degree(according to the start-up cost, interference and some other factors).

If parallelism decreases, the time complexity will increase.

But as we calculated above, the total work complexity is a stable value. The space complexity increases when the parallelism increases, because more memory will be needed for computing.