Map Reduce - Report

1. Highlights

   Used MPI and pthreads.

   a. MPI_Tag

      0: chunkID/map task's ID for the map phase
      1: nodeID for the map phase
      2: reducer task's ID for the reduce phase
      3: nodeID for the reduce phase
      4: finished chunkID for the map phase
      5: nodeID for the map phase, indicating the nodeID has finished all of its tasks
      6: finished chunkID for the map phase, indicating the remaining finished tasks' ID
      7: remote chunkID for map phase

   b. Map Tasker Scheduling

      I am using list from STL to act as a queue to store the map tasks and the map task consists of a pair made of nodeID and chunkID, this is to preserve the data locality. Then for scheduling, when a worker node requests a map task, then the master node will check for the first matching map task with a matching nodeID as the requesting node's rank.

      If there is no more task for the requesting node, then the master node will check if there are still tasks left for other nodes, if there is then it will send a sleep command to the requesting node (the command is represented by chunkID = -2).

   c. Map Function

      Each worker node has its own thread pool, implemented by using pthread, so as long as there are still free threads available, it will request a new map task from the master node. After the worker node receives a map task, it will add the task to its own task queue and the free threads will always check the queue if there is a task available, then it will pop out the first task and execute it.

      As it is complex to send the whole map function output of each chunk to the master node for the next partition stage. I implemented this part by outputting the map function output of each chunk to an output file.

   d. Sorting and Grouping Function

      Since it is said that for the demo, we will be required to modify the sorting and grouping function, I implemented a separate sorting function for ease of modification. Additionally, for the grouping function, I implemented it using the substrate for string, so that I can easily change the initial position to other substrings if it is required to change the group function later.

```
for(auto word: reducerTask){
    //CHANGE GROUP FUNCTION HERE
    string s = word.first.substr(0, string::npos);
    if(group.count(s) == 0) orderedWord.push_back(s);
    group[s].push_back(word.second);
}
```

e. Termination Condition

For the first phase, which is the map phase, the master node will send a termination signal to each of the worker nodes, represented as chunkID = -1. Next, the worker node's threads will stop requesting a new task after receiving the termination signal, and it will wait until every thread has finished its map tasks. The master node will also wait for every unfinished task of each worker node to complete before proceeding to the partition stage.

For the second phase, which is the reduce phase. It is much simpler compared to the previous phase, as there is only one reducer for each of the worker node, so after all of the reduce tasks have been dispatched, the master node just needs to wait for each reducer's request for a new task and send them the termination signal and does not need to wait for the task to finish, because if they request a new task, it means that their last task is finished.

f. Log File

Every output line of the log file is written by the master node. For writing the completion time of a task, there is a map that maps a task to its starting time. Every time a worker node requests a new task, the master node will first record the time and take the difference between the time and the starting time of that task.

For the map phase, since there is more than one task for one worker node. After the termination signal is given to each worker node, each worker node will wait until its last remaining tasks finish before sending all of the finished task IDs back to the master node. This way, the master node will only proceed to the reduce phase after all of the map tasks have finished.

2. Detail of Implementation
    a. Map Phase
        i. Map Function

        For the master node, the implementation is mentioned in the Highlights section above. For the worker node, every one of them will have its own

thread pool with (num of cpus - 1) threads, and the thread pool will terminate, when there are no mapper tasks left. When there are free threads in the thread pool, then the node will request new map tasks, one thread is responsible for one map task, so the node will receive the chunkID from the master node, and if the chunkID is equal to -1, it means that there are no tasks left to do, so it can terminate the thread pool after all of its current tasks have finished. If the chunkID is equal to -2, then the thread has to sleep for the specified time. For sleep time, I am using MPI_Wtime(). Additionally, when requesting new tasks, each node will also send the finished chunkID back to the master node, this is for writing to the log file. When chunkID = -1, then it will wait until all of its current tasks to finish before sending a signal (nodeID) to the master node with MPI_Tag 5, and sending the finished chunkID with MPI_Tag 6, so the master node can write the completion time of each chunkID to the log file.

ii. Partition Function

For this function, a word will be distributed to the reducer with id = [its word hash % num of reducer]. The hash function here uses the hash function from std.

b. Reduce Phase

For this phase, the master node will send reducer tasks to the node requesting new tasks. It will send tasks based on the id order, which is sorted from the smallest to the largest id. Since each node will only have one reducer after all of the reducer tasks have finished, the master node only needs to send the termination signal to each of the reducers when they requested a new task, it does not need to wait like in the previous phase.

i. Sort Function

The default implementation is sorted based on the ASCII character of the word, for ease of modification in the demo, I added a separate sort function.

```
struct {
    bool operator()(pair<string, int> a, pair<string, int> b) const { return a.first < b.first; }
} customSort;
```

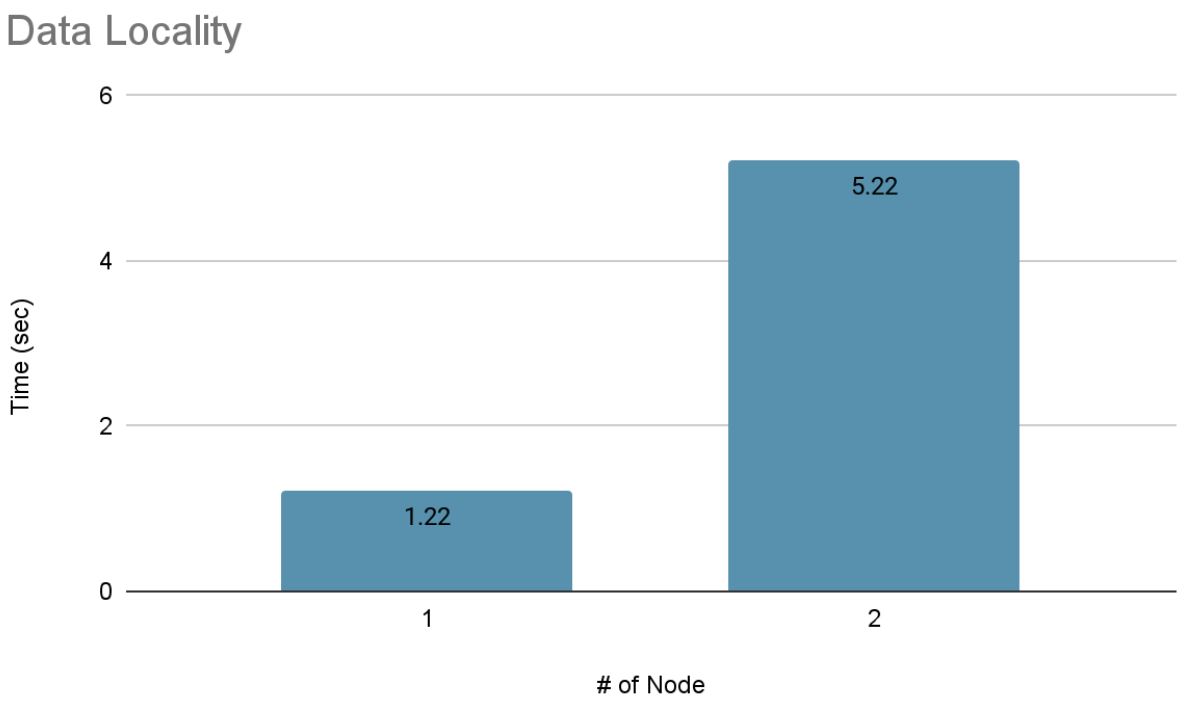
ii. Group Function

The default is grouped by word, which means only the same words can be grouped together. Here, I also used substr method to get the correct grouping substring. There is also a vector for storing the group, and 1 group map for mapping group words to their count. As for why using vectors and maps, it is because I want to preserve the sort order of the group and also map each group with the count of each member of the group.

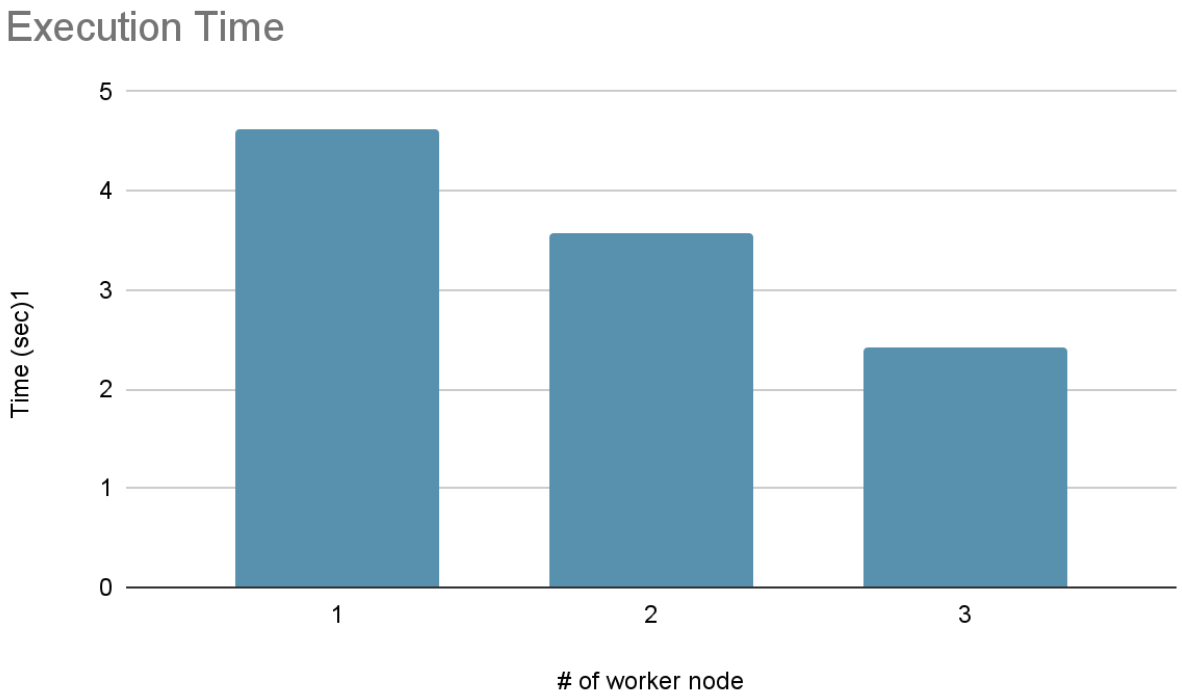
iii. Reduce Function
Iterate every group in the group vector, then iterate every member of the group and add their sum together.

iv. Output Function
Write the output from reduce function to the corresponding output file (based on reducer ID).

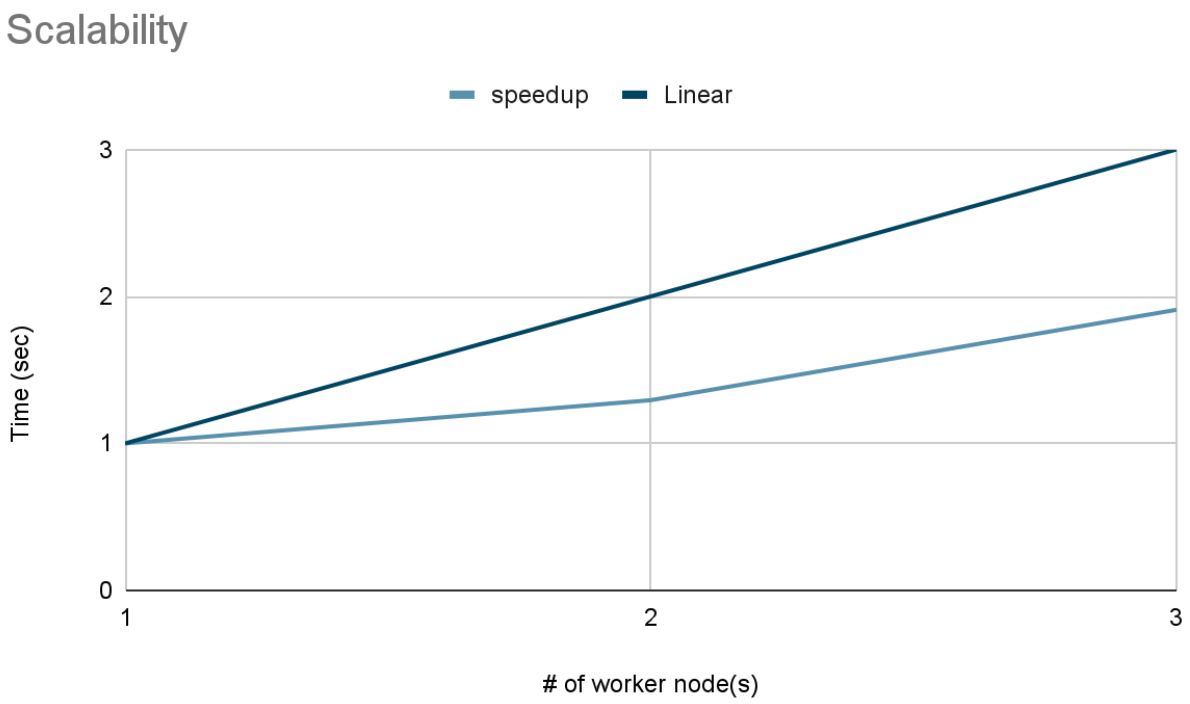
3. Experiments
   a. Data Locality with 2 nodes (Testcase 05 with chunk size 80, 5 chunks)
   Note:
   1: data is distributed to 2 nodes with a ratio 1:1
   2: data is distributed to 2 nodes with a ratio 5:0



   b. Time Profile (testcase 06 with balanced data locality)



c. Scalability (testcase 06 same as above)



Discussion:

For the data locality, the time increases from 1 second to 5 seconds because of the delay sleep time for the node to execute tasks located in another node, so the data locality is really important. For the execution time, since it really depends a lot on the data locality, I distributed the data equally according to the number of worker nodes, and it can be seen from the scalability and execution time figures that the increased number of nodes also decreased the execution time as the tasks executed by each node is decreased.