CS425/ECE428 MP4 Report

Yen-Chieh Sung (ycsung2), Chih-Shin Wang (cswang6)

Design

- Map Task: Workflow Overview
  When a map task is launched, master node will initiate all workers, assign input files to workers, and wait the workers to process all the input files. After an input file is finished by a worker, it will notify the master. When the master sees that all input files are finished, it will multicast all workers to write intermediate files. A worker will pull subsets of produced keys from other workers.
- Map Task Stages
  In our implementation, the whole map task workflow has the following stages:
  1. Workers preparation: In this stage, master will notify all the worker the nodes involving the calculation so that for the following stages each worker can communicate with different workers and the master. In this stage all the worker will also get the executable of the map task from SDFS.
  2. Input files processing: Master node will get all the input file names under given sdfs_src_directory from SDFS. Then, it will assign those files to workers in a round-robin manner. Master node will hold a table recording file assignment for failure handling purpose. Worker will grab input files from SDFS and feed the map task executable with inputs. Each time an input file is done, worker will hold the result and notify master. Also, worker will keep the results and group them by hash value of key of key-value pair. Master node will count the number of completed input files. When all of the files are completed, the cluster will go to next stage.
  3. Writing intermediate file: Master node will multicast all workers to collect results (intermediate key-value) and write intermediate files. When starting to write intermediate file, worker calculate the key range it is responsible for, and pull corresponding results from other workers. Workers will write the intermediate files for each key it collected. After a worker finishes writing all intermediate files it's responsible for, it'll notify the master.
  4. Finishing: Master will tell workers to release the memory of results and reset stateful variables.
- Map Task Failure Handling
  We assume no failure on master node in our design. Stage 1 is fast so we assume no failure during the stage. When failures happens in stage 2, master will look into file assignment table and redispatch the input files assigned to failed nodes to alive workers. When more than one failure happens during stage 3, there may be information lost. Thus the system will roll back to stage 2 and master node will redispatch samely. Failures in stage 4 do not matter since all intermediate files are already uploaded to SDFS.
- Reduce Task: Workflow Overview
  The workflow is similar to map task except that this time only master node is responsible for writing the final result. When a reduce task is launched, master node will assign intermediate files to workers and wait the workers to process all the intermediate files. Workers will send result back to master once an intermediate file is completed. After collection the master will write output file.
- Reduce Task Stages
  In our implementation, the whole reduce task workflow has the following stages:

1. Workers preparation: Master will notify all the worker who is the master so that workers will know who to send back result. In this stage all the worker will also grab the executable of the reduce task from SDFS.
2. Intermediate files processing: Master node will get all the matched intermediate file names with given sdfs_intermediate_filename_prefix from SDFS. Then, it will assign those files to workers in a round-robin manner. Master node will hold a table recording file assignment for failure handling purpose. Worker will grab input files it is assigned from SDFS and feed the reduce task executable with inputs. Each time an intermediate file is done, the worker will send back the result to the master. Master node will count the number of completed intermediate files. When all of the files are completed, the cluster will go to next stage.
3. Writing file: Master node will write final output file by iterate through results it received.
4. Finishing: Master will delete intermediate files if required by user command. Master will also tell workers to release the memory and reset stateful variables.

- Reduce Task Failure Handling
  We assume no failure on master node in our design. Stage 1 is fast so we assume no failure during the stage. When failures happens in stage 2, master will look into file assignment table and redispatch the intermediate files assigned to failed nodes to alive workers. Stage 3 & 4 only have master node involved so there is no need to handle failure.

Speed Up (Optimization)

During experiments we found our system no that scalable. To make it run faster, we do the following improvement.

- Connection Limiting
  Originally we have an RPC call for each dispatched input/intermediate file. However when there are too many files (especially intermediate files), there will be too many connections and leads to 2 issues. The first one is “too many open files,” since there are too many connections at the same time, our main process cannot start new RPC calls for undispatched files. The second issue is that since there are “too many open files,” our heartbeat thread can no longer send any heartbeat and workers start becoming dead (false detection).
- Memory Constraint
  Master node will run out of memory when trying to write large reduced result directly to SDFS. We think it is due to some unknown issue when concatenating results into string in Golang. To solve the problem we write the reduced result line by line directly into local file, and then fork another child process to upload the local file to the SDFS.
  This problem sometimes is because the inefficient garbage collector of Golang
- Thread Number Limiting
  There is still “too many open files” issue when workers are processing input/intermediate files. To solve the problem, we use the Consumer/Producer manner, i.e. create a Map/Reduce request queue and let a fixed number of threads processing the queue, avoiding opening too many files (Golang's channel, download file from SDFS, other RPC connections, etc) at the same time.

Measurement

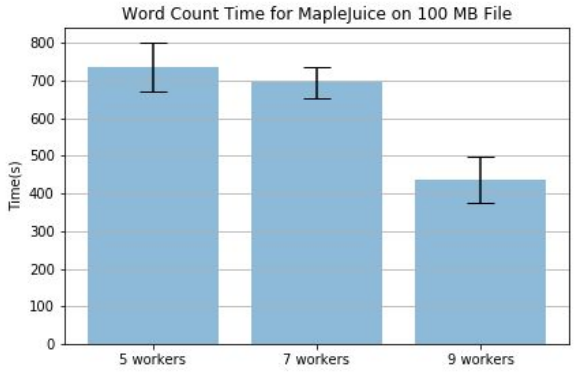
During the measurement we found some VM nodes acting slow so we do not include them as workers. Also, due to the huge runtime needed for large data, we only have a few measurements.

Task 1: Word Count

Dataset: 263 books (.txt) from https://www.gutenberg.org

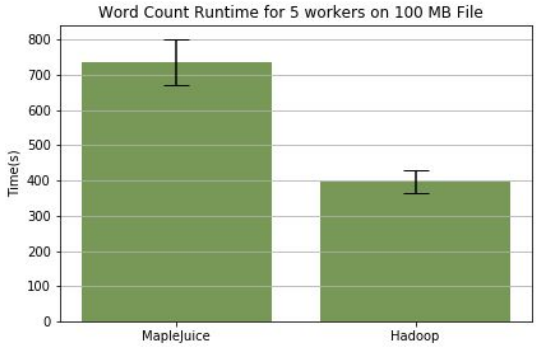
Dataset total size: 106 MB

1. MapleJuice:



Having more workers does not guarantee shorter runtime. First of all, master dispatch files by balancing file number but not exact sum of size of files. Also, files are interleaved, not assigned continuously. Hence, workers' load might coincidently be balance when worker number is small but not balanced when number is higher. For example, assign 6 files with size [1,2,3,1,2,3] to 2 workers, each of them will have 1+2+3=6 unit of file size to process. However, if there are 3 workers, worker 1 will get 1+1=2, worker 2 will have 2+2=4, and worker 3 will have 3+3=6, making loads unbalanced. Second reason of the observation is that our master does not dynamically balance the load. We observed that the performance of nodes fluctuate frequently. When a machine slows down, other nodes will need to wait for that machine, which ends up larger runtime. Thus the statistic might not be accurate without more measurement at different times.

2. Compare with Hadoop (5 machines):



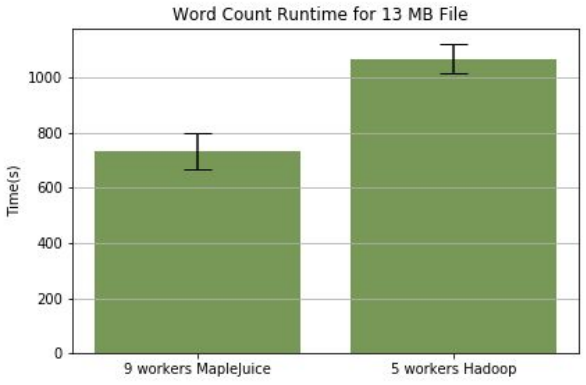
Our system cannot beat Hadoop. We believe that this is related to dynamic load balancing. In addition, our system somehow is not efficient for reading and writing ten to hundreds of intermediate files. Besides, we have our SDFS system and worker share the same process, which makes the worker having less resource to do the worker jobs.

Task 2: Reverse Web-Link Graph

Dataset: http://snap.stanford.edu/data/web-BerkStan.html

Dataset total size: 13 MB

There were 685230 nodes (keys) and 7600595 edges (input key-value pairs). We split the single file randomly to 500 files. Due to network congestion, we only have time to test with portion of the data, reducing to only 90000 nodes (data size 13 MB). This task has more keys, which leads to more intermediate files. We can see that we outperformed hadoop.



However, this is because we have more node (due to network congestion, we are not able to test different # of nodes), hadoop has a more robust resource managing method. For example, our system suffers from memory issues and # of open file issues. When the dataset is small, our system survives and may have chance to outperform hadoop, but when the dataset is larger, then our system downgrades significantly.