NTHU CS542200 Parallel Programming
Homework 4: MapReduce

111062625 蔡哲平

1. Highlights

- Self-implemented thread pool

  Instead of using the thread pool code provided by TAs, I chose to implement the thread pool by myself using mutex locks and condition variables.

2. Implementation

- Process Level

  In this part, we discuss process-level communication. MPI_Send and MPI_Recv are used for all communication between the JobTracker and the TaskTrackers. Since there are lots of communication among them, the value of the tag is set to a different value to distinguish different purposes. The communication of the JobTracker distributing tasks to the TaskTrackers can be divided into the following phases:

  - Phase 1: JobTracker receiving task requests from the TaskTrackers

    In this phase, the JobTracker will first use MPI_Recv to get the request from the TaskTrackers. The source is set to MPI_ANY_SOURCE to get the request from an arbitrary TaskTracker. The TaskTracker will send its node_id to the JobTracker. Once the JobTracker gets the node_id, it will send the task to the corresponding node. This phase will keep running until the task pool is empty.

  - Phase 2: JobTracker notifies the TaskTrackers that there are no more tasks

    In this phase, the JobTracker will first use MPI_Recv to receive the last requests from all TaskTrackers. Next, it will send -1 to all TaskTrackers. Once the TaskTracker gets -1, it will know that the task pool is empty.

  - Phase 3: TaskTrackers send elapsed time of each task back to the JobTracker

    Once the TaskTracker gets -1 from the JobTracker, it will send back the elapsed time of each task when the task is finished.

- Thread Level

  In this part, we discuss thread-level communication. Since I created a thread pool to automatically get a new task once the thread becomes available, the following things should be aware of:

  - Mutex lock

    A mutex lock is used to make sure that there is only one thread in the critical

section at each moment. In my implementation, the task queue inside each node must be protected by a mutex lock since a simultaneous push pop of different threads may cause a race condition. Another data that must be protected is num_working, which records the number of the current working threads.

- Condition variables

  A condition variable is used for a thread to keep waiting until an event is signaled. In my implementation, a thread needs to wait until the task queue isn't empty to pop a task from the task queue. This event will be signaled by the node each time it pushes a new task in. Moreover, the node should also wait until there are available threads to compute a task to push a new task into the task queue. This event will be signaled each when a thread finishes its task.

- Intermediate Result & Shuffle

  In my implementation, once a mapping task is finished, it will generate an intermediate result file. Hence, the number of intermediate result files is the same as the number of mapping tasks, which also equals the number of chunks. After all mapping tasks are finished, the JobTracker will create NUM_REDUCER files to store the shuffled results. Next, the JobTracker will read all contents in the intermediate result files and partition the key. Finally, write the key-value pair to the file according to the partitioned result.

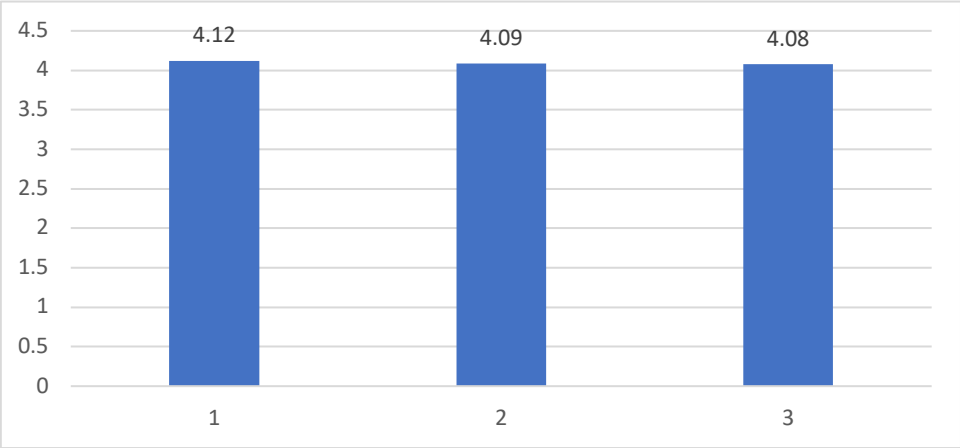
3. Experiment & Analysis

i. System spec

All experiments are conducted on the Apollo server. Test case's parameters: NODES=4, CPUS=6, NUM_REDUCER=10, DELAY=4, NUM_CHUNK=30, CHUNK_SIZE=320.

ii. Data Locality

- Experiment 1: Observe how the data locality affects the runtime. The value of the X axis denotes how many nodes are used for data locality, e.g., if the value is 2, it means that data are evenly distributed on 2 nodes.

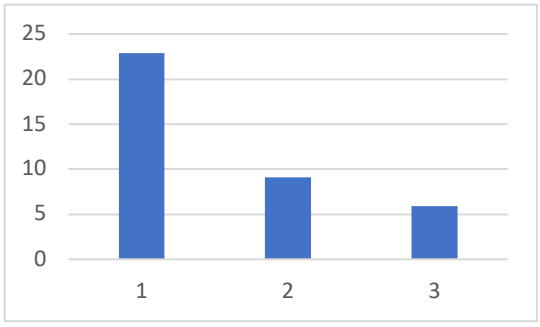


Runtime of different data locality
[X: # nodes for data locality, Y: Runtime (sec)]

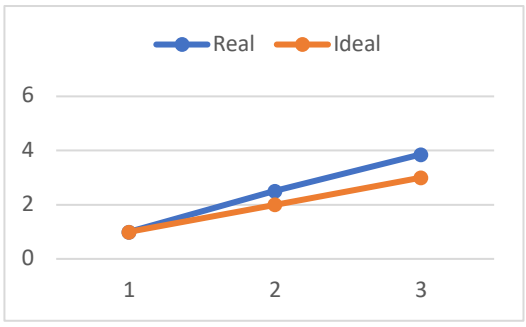
Observation 1: According to the above figure, we can see that the runtime of a different number of nodes for data locality is almost the same as the delay time. In my opinion, this is because those nodes with data locality can finish the task in an extremely short runtime. After nodes that don't have data locality wake up because of the delay, all tasks are already finished by other nodes.

iii. Scalability

- Experiment 2: Observe how the number of nodes affects the runtime. Note that the data are evenly distributed on nodes.



Runtime of different # of nodes
[X: # of nodes, Y: Runtime (sec)]



Scalability of different # of nodes
[X: # of nodes, Y: Scalability)]

- Observation 2: According to the above figure, we can see that as the # of nodes decreases, the runtime decreases too. Moreover, the actual scalability is even better than the ideal scalability. In my opinion, it's because using more nodes not only increases the parallelism but also benefits more from the data locality.

4. Experiences & Conclusion

In this assignment, I implemented a thread pool using mutex locks and condition variables on my own, which gave me a huge sense of accomplishment. Moreover,

implementing the MapReduce framework from scratch is also quite challenging. Using MPI and Pthread at the same time makes it extremely difficult to debug. I found it very important to use different tags when calling MPI_Send and MPI_Recv since I was stuck at a stupid bug for a long time because of this problem. In a nutshell, this is the last assignment of this course, although these assignments are very heavy-loading, I didn't regret taking this course since I've learned a lot from it.