CS425 MP4 Report

Group 4: Yi-Hsin Chen(yihsinc2), Che-Lin Huang (clhuang2)

Design

Master Failure

To deal with master failure, we have two assumptions: **client doesn't fail** and **workers don't fail**. Under our design, our system **could finish the graph computing task** even the original master fails. The key idea is using client/worker to help backup master restore the progress of the task.

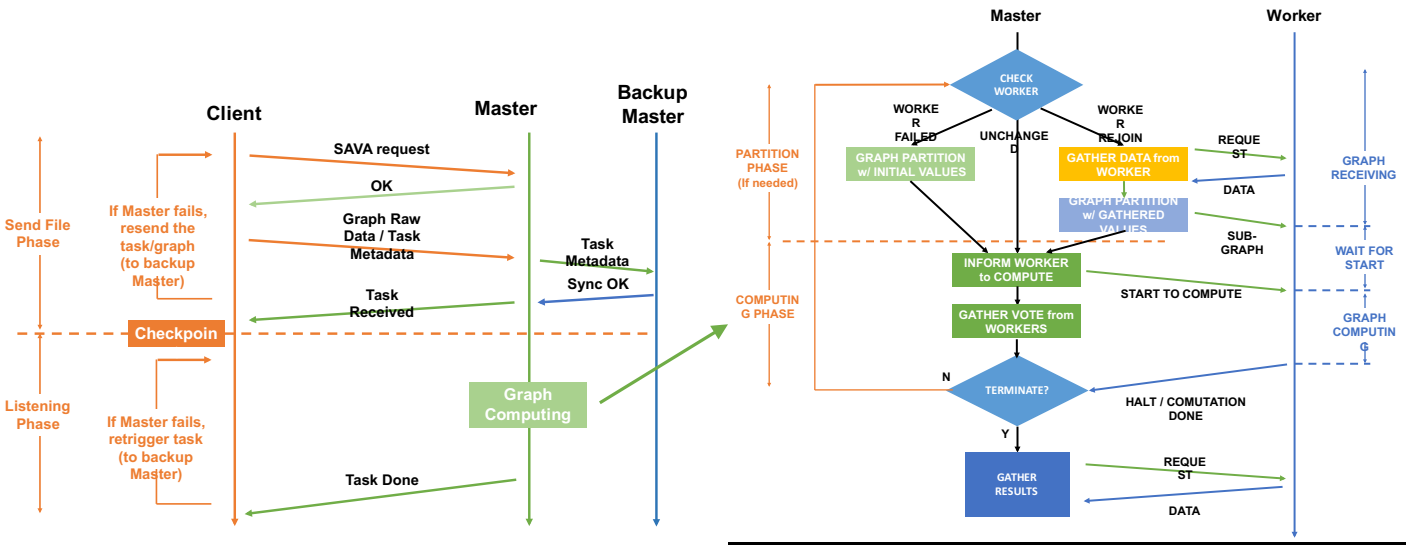
On the client side: once a client sends out a request for graph computing, it will first send the graph to the master. After the maser received the graph raw data and save it into the SDFS, it will synchronize task-related information with backup master. After the synchronization is done, the master will reply a "OK" message to client. This is the check point at the client side, if the master fails before the checkpoint, the client will simply resend task request/graph to the backup master. If the master fails after the checkpoint, since backup master has already gotten the task information (sent by the original master), the client will only **re-trigger** the task (w/o sending graph again).

On the worker side: in each iteration, we could divide the worker status into two phases: **GRAPH RECEIVING / GRAPH COMPUTING**. Once the backup master is up and re-triggered by the client, it will ask workers for their status to decide its action.

1. GRAPH RECEIVING: if master fails before each worker receives the EOF message from the master, workers will send a message to the backup master to ask for re-distribution of the graph.
2. GRAPH COMPUTING: if each worker successfully receives its sub-graph, after computation of the current iteration is completed, each worker will send the current iteration number to the backup master, and the backup master only needs to update its iteration number. And the task could go on.

Worker Rejoins/Fails

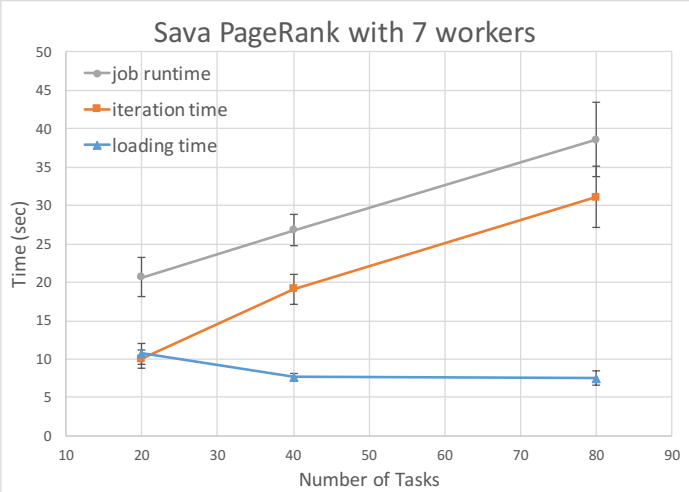
Before the start of each iteration, master will check the worker pool. If at least one worker fails, it simply re-partition the graph and re-distribute it among workers. If at least one new worker joins the cluster, the master will collect the computation results from existing workers first and then re-distribute the graph among the new pool of workers.

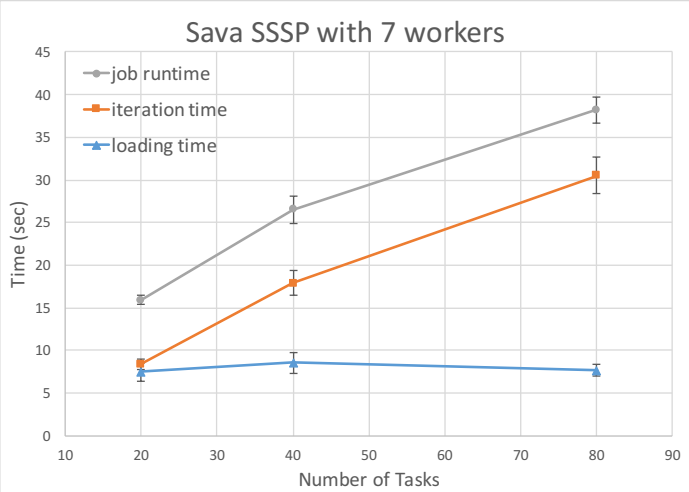


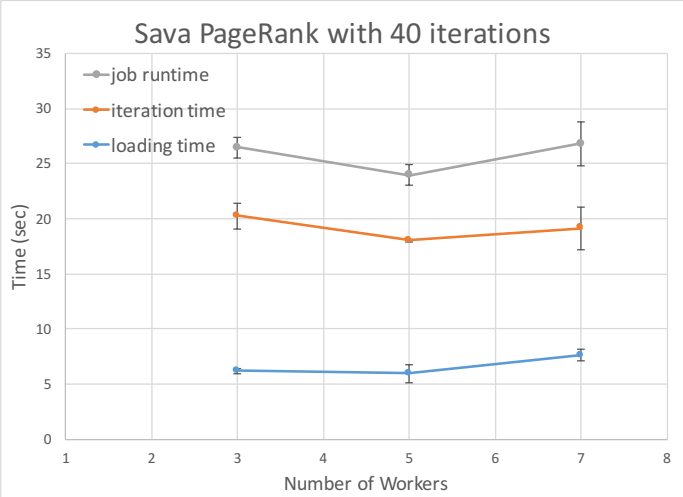
Graph Processing at Workers

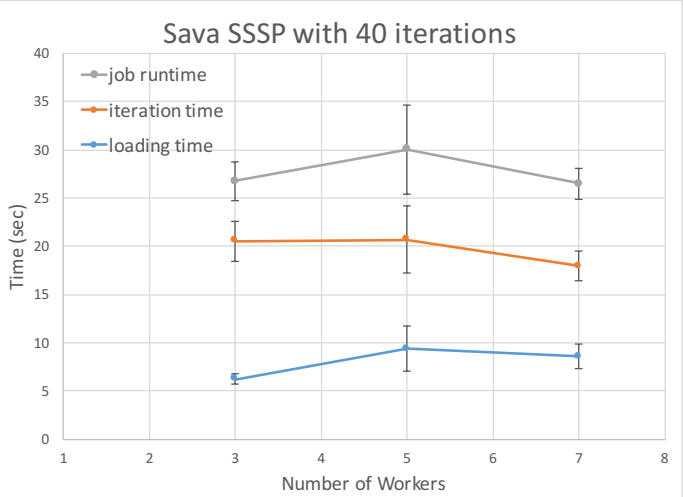
The graph processing workers use Gather-Apply-Scatter model to process the graph related tasks. At each iteration, each work gathers the cached information from the previous iteration, applies the function that is corresponding to the application, and then multithreading scatters the vertexes' value out to their successors at other workers. They also read commands from the master, execute the corresponding command and make sure that it response proper message to the new master upon master fail.

Experiment

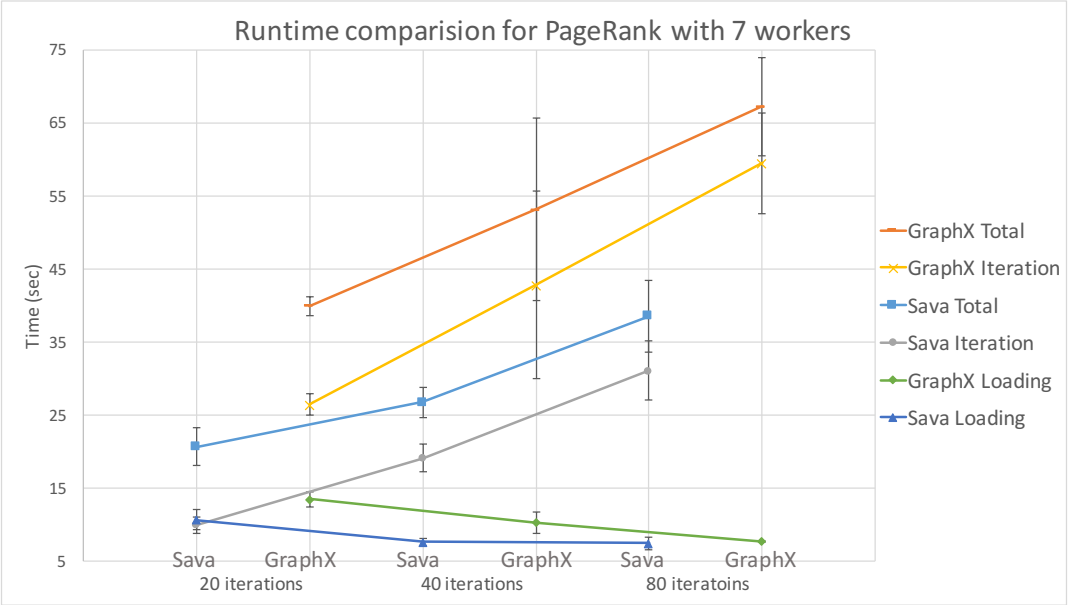


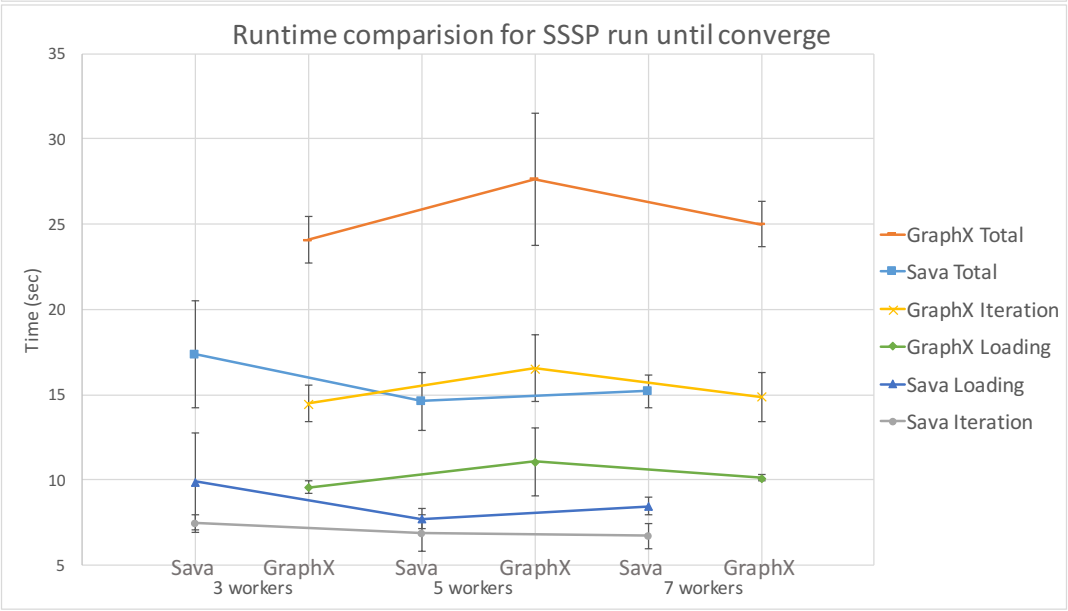






For the experiment, we use Amazon product co-purchasing network with 330k vertices. The plots above show the average and standard deviation of job runtime, iteration time and loading time with respect to different applications, number of tasks and number of workers. The average job runtime for both PageRank and SSSP doesn't vary a lot, which make sense here since none of them require heavying graph processing work. Increasing the number of workers has no significant effect on the runtime. To investigate this phenomenon, we decouple the runtime of Gather-Apply-Scatter stages at the worker, find out that for each iteration at the worker, scattering takes almost 90~95% of the iteration time. For more workers, the gathering time and applying time could significantly reduce. However, the scattering time doesn't have much improvement and this is the reason why given more worker has no significant effect on the runtime. With a fixed number of workers, the iteration time is proportional to the number of iterations and the loading time is about the same regarding the number of iterations, which an expecting result here.





Yes, we beat Spark GraphX in both applications. For the experiment, we use Amazon product co-purchasing network with 330k vertices. The plots above show the average and standard deviation of job runtime, iteration time and loading time. Spark GraphX SSSP couldn't set the number of tasks, so we compare them among the different number of workers. For PageRank we compare them among the different number of iterations. Sava and Spark GraphX have about the same amount of loading time. Where the Sava outperform Spark GraphX is the iteration time. To reason why, we list some difference between Sava and Spark GraphX:

1. Spark platform supports various kind of tasks, while the Sava only accepts graph processing tasks. For the highly specialized application, there are always some places we could do optimization and boost the performance.

2. As soon as we figured out that the transmission takes a tremendous amount of time, we wrote our object serialization and deserialization to reduce the size of the object that needs to transmit between masters and workers. Otherwise, our iteration time only beat spark a little in the first place.